BBM 102 – Introduction to Programming II

Spring 2018

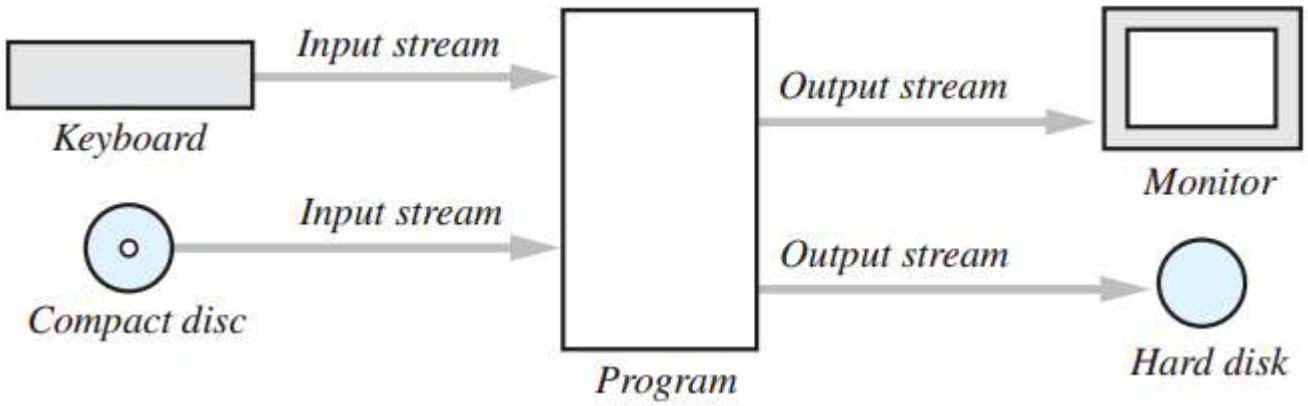
Streams and Input/Output

Today

- Streams and Files
- Text/Binary Files
- java.io.File class
- Revisiting java.util.Scanner
- Java I/O Library
- Decorator Pattern
- **InputStream**s and **OutputStream**s
- **Reader**s and **Writer**s
- Sequential Access vs Random Access
- java.io.RandomAccessFile
- Serialization

Streams

- A **stream** is a flow of data. The data might be characters, numbers, or bytes consisting of binary digits.
- If the data flows into your program, the stream is called an **input stream** (example: **System.in**).
- If the data flows out of your program, the stream is called an **output stream** (example: **System.out**).



Files

- The keyboard and the screen deal with temporary data
- Files provide a way to store data permanently
- All of the data in any file is stored as bits, or 0s and 1s.
- Files are categorized as **text files** and **binary files**

Text/Binary Files

- Text files
  - The bits represent printable (easily readable by humans when printed) characters.
  - The characters are coded with a "character set", ASCII, ISO-8859-1, utf-8..
  - They can be edited with a " text editor "
  - Examples: Program source files (.java, .c), files saved with a text editor, e.g. *Notepad.exe*
- Binary Files
  - The bits represent other types of encoded information, such as executable instructions or numeric data
  - They are easily read by the computer but not humans
  - They are *not* "printable" files
  - Examples: Executables (.exe), images (.jpg, .png), music (.mp3), or video (.avi, .mov) files

ASCII (American Standard Code for Information Interchange) Code Table

Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH	(start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX	(start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX	(end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT	(end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ	(enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK	(acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL	(bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS	(backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB	(horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT	(vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR	(carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO	(shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI	(shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE	(data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1	(device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2	(device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3	(device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4	(device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK	(negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN	(synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB	(end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN	(cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM	(end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB	(substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC	(escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS	(file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS	(group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS	(record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US	(unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Extended ASCII Codes

128	Ç	144	É	160	á	176	░	192	└	208	╨	224	α	240	≡
129	ü	145	æ	161	í	177	▒	193	┴	209	╤	225	ß	241	±
130	é	146	Æ	162	ó	178	▓	194	┬	210	╥	226	Γ	242	≥
131	â	147	ô	163	ú	179	│	195	├	211	╙	227	π	243	≤
132	ä	148	ö	164	ñ	180	┤	196	─	212	╘	228	Σ	244	⌠
133	à	149	ò	165	Ñ	181	╡	197	┼	213	╒	229	σ	245	⌡
134	å	150	û	166	ª	182	╢	198	╞	214	╓	230	µ	246	÷
135	ç	151	ù	167	º	183	╖	199	╟	215	╫	231	τ	247	≈
136	ê	152	ÿ	168	¿	184	╕	200	╚	216	╪	232	Φ	248	°
137	ë	153	Ö	169	⌐	185	╣	201	╔	217	┘	233	Θ	249	∙
138	è	154	Ü	170	¬	186	║	202	╩	218	┌	234	Ω	250	·
139	ï	155	¢	171	½	187	╗	203	╦	219	█	235	δ	251	√
140	î	156	£	172	¼	188	╝	204	╠	220	▄	236	∞	252	ⁿ
141	ì	157	¥	173	¡	189	╜	205	═	221	▌	237	φ	253	²
142	Ä	158	₧	174	«	190	╛	206	╬	222	▐	238	ε	254	■
143	Å	159	ƒ	175	»	191	┐	207	╧	223	▀	239	∩	255	

Source: www.LookupTables.com

www.LookupTables.com

Text/Binary Files

- Confused? Let's see an example: We want to write the number 127 into a file.

- If we write it into an ASCII coded text file:
  - Three bytes will be used for each character: 1 , 2, and 7
  - Binary values of these characters: 00110001, 00110010, 00110111

- If we write it into a binary file:
  - One byte (variable is defined as byte): 01111111
  - Two bytes (variable is defined as short): 00000000 01111111
  - Four bytes (variable is defined as int):
    00000000 00000000 00000000 01111111

java.io.File

- Do not be deceived with the name of it! Class represents a path rather than a file!
- Can be used to
  - Check if the path exists or not
  - Check if the path is a file or a directory
  - Check/edit the file/directory's readable, writable, executable, hidden properties
  - Create/delete file/directory
  - Get the contents of a directory
  - Get the last modification date and time of the file/directory

FileExample Program

```
public class FileExample {
    public static void main(String[] args) {
        File path = new File("h:\\example");
        if (!path.exists()) {                    // It does not exist, create a directory!
            path.mkdir();
        } else if (path.isDirectory()) { // It is a directory! List the contents
            String[] contentOfDirectory = path.list();
            for (String filename : contentOfDirectory) {
                System.out.println(filename);
            }
        } else {                                 // It is a file! Display the properties of the file
            System.out.println("Read:" + path.canRead() +
                ", Write: " + path.canWrite() + ", Hidden: " + path.isHidden());
        }
    }
}
```

Revisiting java.util.Scanner

- Class Scanner is an easy way to read input from keyboard, remember?

```
// create a scanner System.in (keyboard)
Scanner scanner = new Scanner(System.in);
// read a string from keyboard and write it to System.out (monitor)
System.out.println(scanner.next());
```

- It takes an inputstream to its constructor and reads from it

- What if we give a File object to the constructor?

```
// create a scanner for the file example.txt
scanner = new Scanner(new File("c:example.txt"));
// read a string from the file and write it to System.out (monitor)
System.out.println(scanner.next());
```

Scanner example: display contents of a file

```
public static void main(String[] args) {
        Scanner scanner = null;
        try {
                scanner = new Scanner(new File(args[0]));
                while (scanner.hasNext()) {
                        System.out.println(scanner.nextLine());
                }
        } catch (Exception e) {
                e.printStackTrace();
        } finally {
                if (scanner != null) scanner.close();
        }
}
```

Java I/O Library

- Mostly under the package **java.io**
- Includes classes, interfaces and exceptions for
  - Input/Output
  - Binary/Text
  - Sequential/Random Access
- JDK versions improved the library in time, adding new classes/interfaces.

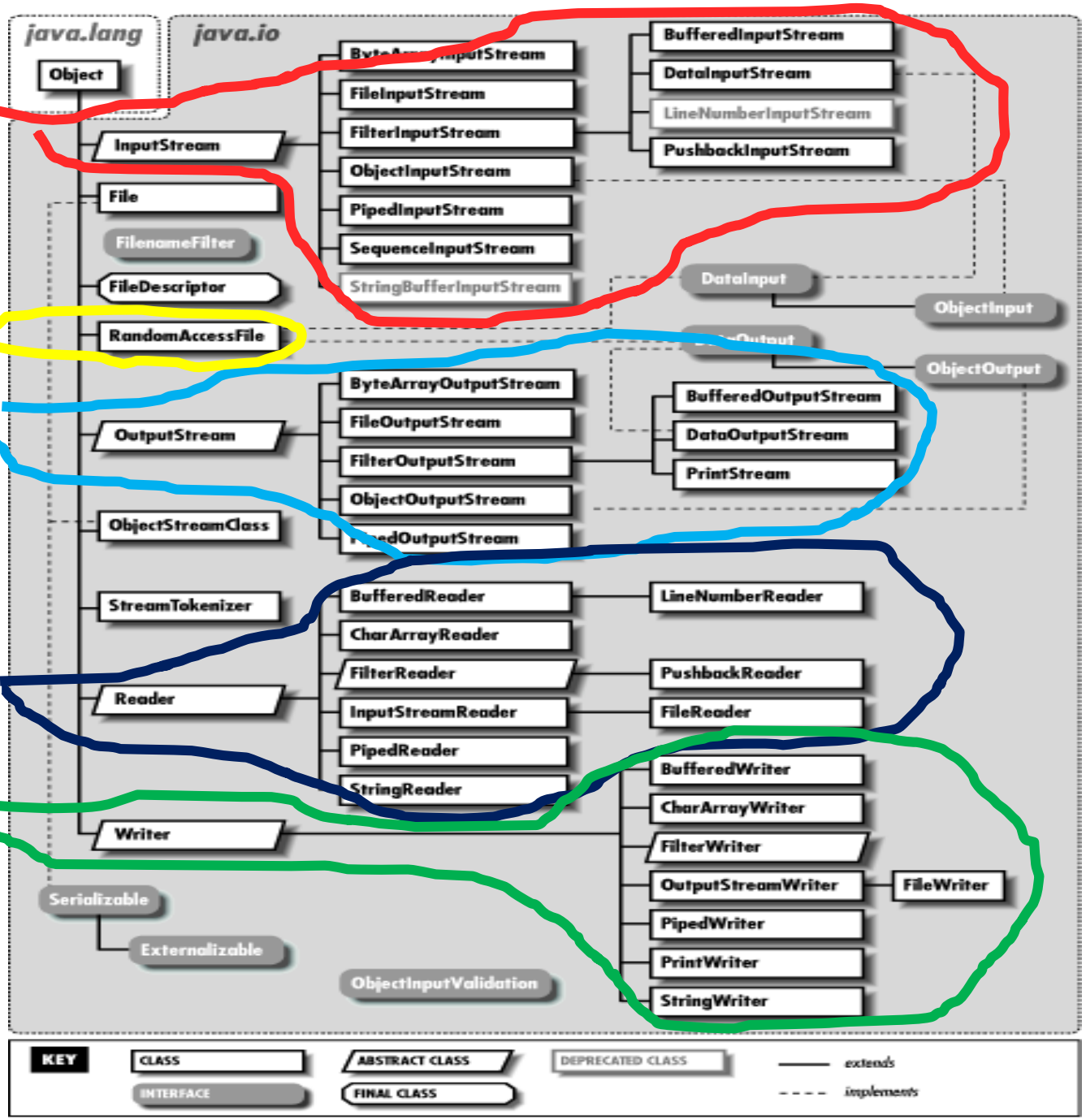
Binary Input
(byte oriented)

Random Access

Binary Output
(byte oriented)

Text Input
(character oriented)

Text Output
(character oriented)



Creating a text file

- An easy way to create a text file is using **java.io.PrintWriter**

```
public static void main(String[] args) {
    PrintWriter outputStream = null;
    try {
        outputStream = new PrintWriter("c:out.txt");  // open the file
        outputStream.println("Example line.."); // write something to the file
    } catch(FileNotFoundException e) {
        System.out.println("Error opening the file!");
    } finally {
        if (outputStream != null)    outputStream.close(); // close the file
    }
}
```

Example: from keyboard to file

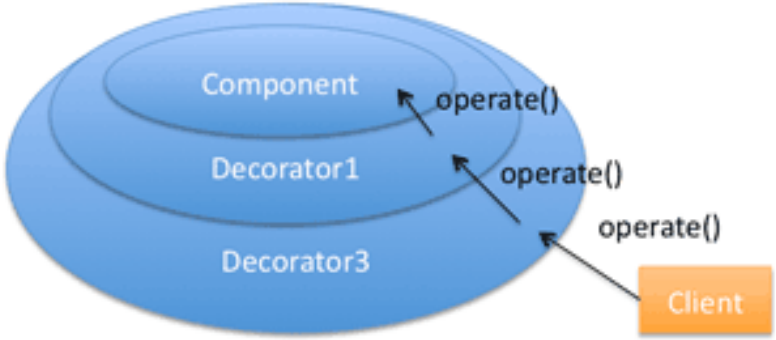
```
public static void main(String[] args) {
    PrintWriter outputStream = null;
    Scanner scanner = null;
    try {
        outputStream = new PrintWriter(args[0]);   // open the file
        scanner = new Scanner(System.in);          // create scanner for keyboard
        String str = scanner.nextLine();           // get the first line
        while (!str.equalsIgnoreCase("exit")) {    // if it is not «exit»
            outputStream.println(str);             // write it to the file
            str = scanner.nextLine();              // get a new line
        }
    } catch(FileNotFoundException e) {
        System.out.println("Error opening the file!");
    } finally {
        if (outputStream != null) outputStream.close();     // close the file
        if (scanner != null) scanner.close();               // close the scanner
    }
}
```

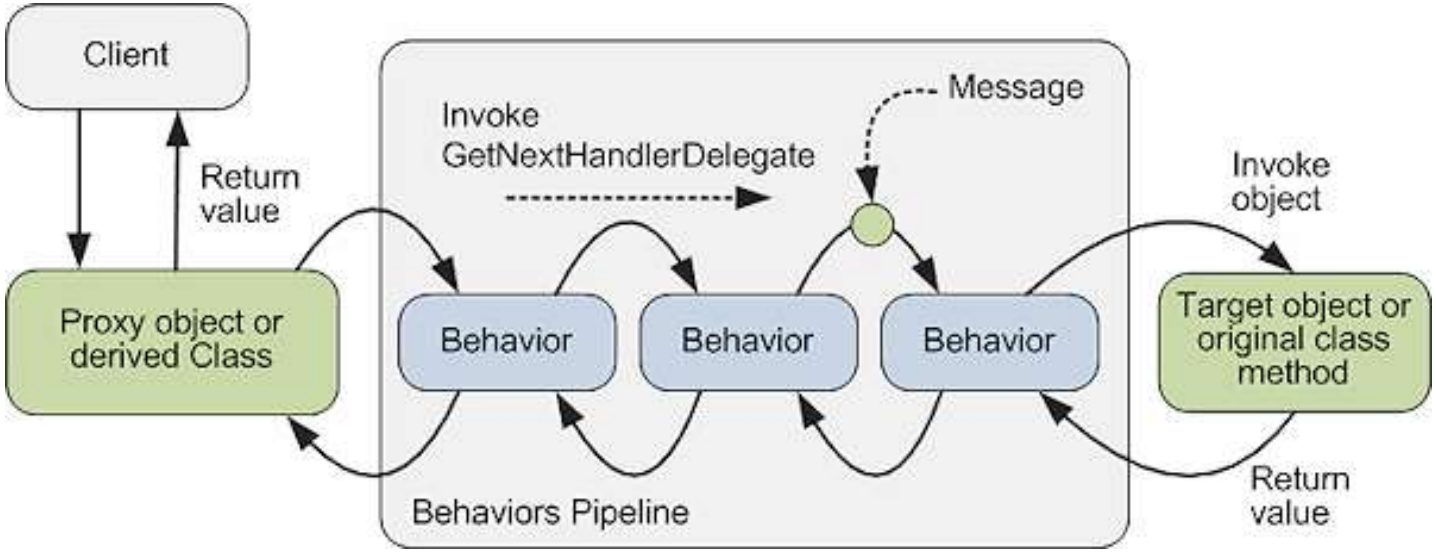
Decorator Pattern

- Software Design Patterns
  - "In software engineering, a **design pattern** is a general reusable solution to a commonly occurring problem within a given context in software design" (wikipedia)
  - Design patterns gained popularity in computer science after the book ***Design Patterns: Elements of Reusable Object-Oriented Software*** was published in 1994 by the so-called "Gang of Four" (Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides), which is frequently abbreviated as ”GoF".

Decorator Pattern

- Decorator Pattern adds a new functionality to an existing object
- A decorator class decorates an inner object and uses its methods to serve in a different way





Decorator Pattern in java.io

- **InputStream** and **Reader** classes (and their subclasses) has basic methods called **read()** for reading a single byte or an array of bytes
- **OutputStream** and **Writer** classes (and their subclasses) has basic methods called **write()** for writing a single byte or an array of bytes
- Problem: A new access to the disk for each byte will slow down the application seriously
- Solution: Bytes may be collected before reading from or writing to the disk. This will reduce the number of physical disk operations
- Decorator classes
  - java.io.BufferedInputStream, java.io.BufferedReader
  - java.io.BufferedOutputStream, java.io.BufferedWriter

BufferedReader example

```
public static void main(String[] args) {
    BufferedReader reader = null;
    try {
        reader = new BufferedReader(new FileReader(new File(args[0])));
        String line;
        while ((line = reader.readLine()) != null) {
            System.out.println(line);
        }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        if (reader != null) reader.close();
    }
}
```

A more complicated decoration example

- Let's say that we have a bunch of Java objects in a Gzipped file named ‘objects.gz’ and that we want to read them a bit quickly

```
// First open an inputstream of it:
FileInputStream fis = new FileInputStream("objects.gz");
// We want speeeed, so let's buffer it in memory:
BufferedInputStream bis = new BufferedInputStream(fis);
// The file is gzipped, so we need to ungzip it:
GzipInputStream gis = new GzipInputStream(bis);
// We need to read those Java objects:
ObjectInputStream ois = new ObjectInputStream(gis);
// Now we can finally use it:
SomeObject someObject = (SomeObject) ois.readObject();
```

InputStream and subclasses

InputStream's job is to represent classes that produce input from different sources. These sources can be:

- An array of bytes (**java.io.ByteArrayInputStream**)
- A String object (**java.io.StringBufferInputStream**)
- A file (**java.io.FileInputStream**)
- A "pipe," (**java.io.PipedInputStream**)
  - Pipe works like a physical pipe: You put things in at one end and they come out the other.
- A sequence of other streams, so you can collect them together into a single stream (**java.io.SequenceInputStream**)
- Other sources, such as an Internet connection

OutputStream and subclasses

- An array of bytes (**java.io.ByteArrayOutputStream**)
- A file (**java.io.FileOutputStream**)
- A "pipe," (**java.io.PipedOutputStream**)
  - Pipe works like a physical pipe: You put things in at one end and they come out the other.

Homework

- Go over the input and out stream classes mentioned in the previous two slides!
- Try to understand at least how they basically work.

Decorating InputStreams

- **java.io.DataInputStream**: read primitives (int, char, long, etc.) from a stream in a portable fashion.

- **java.io.BufferedInputStream**: prevents a physical read every time you want more data.

- **java.io.LineNumberInputStream**: Keeps track of line numbers in the input stream; you can call getLineNumber( ) and setLineNumber (int).
  - *This class incorrectly assumes that bytes adequately represent characters.*

- **java.io.PushbackInputStream**: Has a one-byte pushback buffer so that you can push back the last character read.

Decorating OutputStreams

- **java.io.DataOutputStream**: write primitives (int, char, long, etc.) from a stream in a portable fashion.

- **java.io.BufferedOutputStream**: prevent a physical write every time you send a piece of data.

- **java.io.PrintStream**: For producing formatted output. While DataOutputStream handles the storage of data, PrintStream handles display

Example Program: create a copy of a file

```
public static void main(String[] args) throws Exception {
    BufferedInputStream bis = null;
    BufferedOutputStream bos = null;
    try {
        bis = new BufferedInputStream(new FileInputStream(new File(args[0])));
        bos = new BufferedOutputStream(new FileOutputStream(new File(args[1])));
        byte oneByte;
        // read a byte. -1 will be returned at the end of the file.
        while ((oneByte = bis.read()) != -1) {
            bos.write(oneByte);                    //  write the byte to the output
        }
    } finally {
        if (bis != null) bis.close();              // close the streams
        if (bos != null) bos.close();
    }
}
```

Is it too slow?

```
public static void main(String[] args) throws Exception {
    BufferedInputStream bis = null;
    BufferedOutputStream bos = null;
    byte[] bytes = new byte[1024 * 16];   // bytes will be read in this by 16K chunks
    try {
        bis = new BufferedInputStream(new FileInputStream(new File(args[0])));
        bos = new BufferedOutputStream(new FileOutputStream(new File(args[1])));
        int size;
        while ((size = bis.read(bytes)) > -1) {
            bos.write(bytes);
        }
    } finally {
        if (bis != null) bis.close();
        if (bos != null) bos.close();
    }
}
```

Another example: download a web page

// please note that exception handling is not coded properly!!

```
public static void main(String[] args) throws Exception {
    URL url = new URL("http://web.cs.hacettepe.edu.tr/~bbm102/");
    BufferedInputStream bis = new BufferedInputStream(url.openStream());
    BufferedOutputStream bos = new BufferedOutputStream(
                    new FileOutputStream(new File("downloadedPage.html")));
    for (int c = bis.read(); c != -1; c = bis.read()) {
        bos.write(c);
    }
    bis.close();
    bos.close();
}
```

Readers and Writers

- **InputStream** and **OutputStream** classes provide functionality in the form of **byte oriented I/O**
- **Reader** and **Writer** were added to the library with Java 1.1. These classes provide **Unicode-compliant, character-based I/O**
- Almost all of the original Java I/O stream classes have corresponding **Reader** and **Writer** classes

InputStream/OutputStream Reader/Writer correspondings

InputStream/OutputStream	Reader/Writer
InputStream	Reader adapter: InputStreamReader
OutputStream	Writer adapter: OutputStreamWriter
FileInputStream	FileReader
FileOutputStream	FileWriter
StringBufferInputStream	StringReader
(no corresponding class)	StringWriter
ByteArrayInputStream	CharArrayReader
ByteArrayOutputStream	CharArrayWriter
PipedInputStream	PipedReader
PipedOutputStream	PipedWriter

Decorator correspondings

InputStream/OutputStream	Reader/Writer
BufferedInputStream	BufferedReader
BufferedOutputStream	BufferedWriter
PrintStream	PrintWriter
LineNumberInputStream	LineNumberReader
PushbackInputStream	PushbackReader

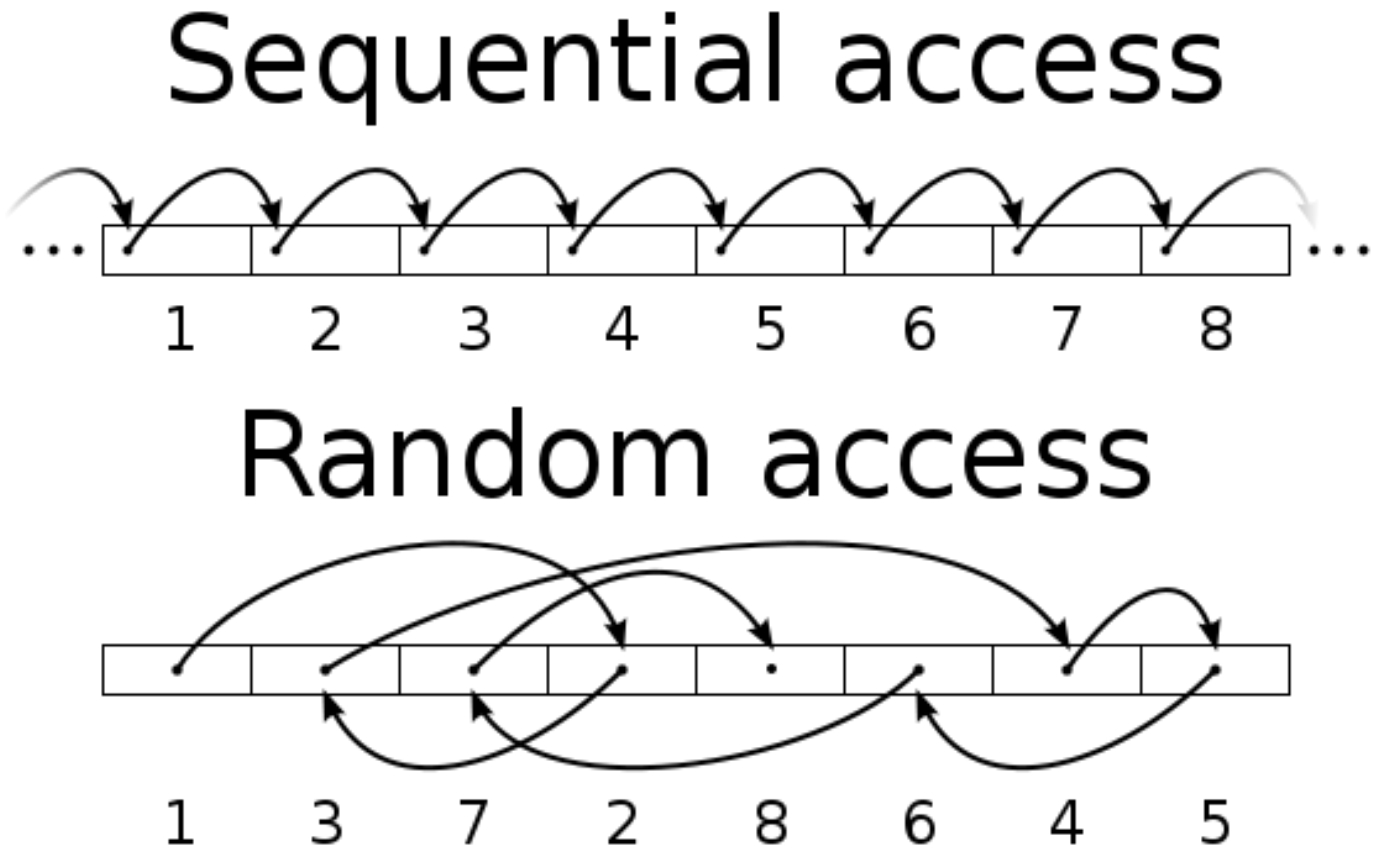
Example program: copy a file line by line

```
public static void main(String[] args) throws Exception {
    BufferedReader br = null;
    PrintWriter pw = null;
    try {
        br = new BufferedReader(new FileReader(new File(args[0])));
        pw = new PrintWriter(
                new BufferedWriter(new FileWriter(new File(args[1]))));
        String line;
        while ((line = br.readLine()) != null) {    // read a line. null at the end of the file
            pw.println(line);                       // write a line
        }
    } finally {
        if (br != null) br.close();
        if (pw != null) pw.close();
    }
}
```

Random Access

- Reading the next byte/string/number or writing to the next location is called sequential access.
- Sequential access is easy and efficient when you don’t know the contents of a file or just want to create a copy of it for example.
- On the other hand, if you know the sizes of records in a file, you can move in the file to read or change a specific record. This is random access.
- All records don’t have to be the same size; you just have to determine how big they are and where they are placed in the file.

Let’s clarify it by an image



java.io.RandomAccessFile

- Used for random access.
- Is not part of the InputStream or OutputStream hierarchy. It's a completely separate class, written from scratch.
- Some methods:
  - getFilePointer( ): find out where you are in the file
  - seek( ): move to a new point in the file
  - length(): return the length of the file
- the constructors require a second argument (identical to fopen() in C) indicating whether you are just randomly reading ("r") or reading and writing ("rw"). There's no support for write-only files

Example program: Editing courses

Course.java

```
public class Course {
    private String code;
    private String name;
    private int credit;

    public Course(String c, String n, int cr) {
        this.setCode(c);
        this.setName(n);
        this.credit = cr;
    }
    public int getCredit() { return credit; }
    public void setCredit(int c) { this.credit = c; }

    public String getCode() { return code; }
    public void setCode(String code) {
        this.code = to40Chars(code);
    }
    public String getName() { return name; }
    public void setName(String name) {
        this.name = to40Chars(name);
    }
    private String to40Chars(String str) {
        String tmp = str;
        for (int i = str.length(); i < 40; i++) {
            tmp += ' ';
        }
        return tmp.substring(0, 40);
    }
    public String toString() {
        return code + " - " +
            name + " - " +
            credit;
    }
}
```

Program.java

RandomAccessFile is given a mode while opening the file. rw: read/write (similar to c)

```
public static void main(String[] args) throws Exception {
    // create course objects
    Course[] courses = new Course[4];
    courses[0] = new Course("BBM101", "Programlamaya Giris I", 3);
    courses[1] = new Course("BBM102", "Programlamaya Giris II", 3);
    courses[2] = new Course("BBM103", "Programlamaya Giris Lab I", 3);
    courses[3] = new Course("BBM104", "Programlamaya Giris Lab II", 3);
    // open the file. It will be accessed randomly
    RandomAccessFile raf = new RandomAccessFile(new File("courses.txt"), "rw");
    // write the courses to a file. Each course is a record
    for (int i = 0; i < courses.length; i++) {
        raf.writeBytes(courses[i].getCode());    // write the code as string
        raf.writeBytes(courses[i].getName());    // write the name as string
        raf.writeInt(courses[i].getCredit());    // write the credit as int
    }
```

Program.java (continued)

```
// let's read the second course's data and create a course object
byte[] bytes = new byte[40];          // data will be read in this as chunks of 40 bytes

// seek to the 2nd record. each record is 40 + 40 + 4 bytes long.
raf.seek((40 + 40 + 4) * (2 - 1));
```

n – 1 records must be skipped to seek to the nth record

credit

name

code

```
raf.read(bytes);
String code = new String(bytes);      // first 40 byte is the code of the course

raf.read(bytes);                      // second 40 byte is the name of the course
String name = new String(bytes);

raf.read(bytes, 0, 4);                // read 4 bytes: the credit
int credit = ByteBuffer.wrap(bytes).getInt();          // convert byte array to int
System.out.println(new Course(code, name, credit));  // create and print the course
```

Program.java

```
    // let's update the name of the 4th course

    // seek to the beginning of name of 4th course
    raf.seek((40 + 40 + 4) * (4 - 1) + 40);

    // write the new name of the course
    raf.writeBytes("Programlamaya Giris Laboratory II");

    // close the file
    raf.close();
} // end of main
```

Seek to the name field

Seek to the 4th record

Homework

- Investigate file opening modes in Java!
  - read, write, append, …

Serialization

- "**Serialization** is the process of translating data structures or object state into a format that can be stored (for example, in a file or memory buffer, or transmitted across a network connection link) and reconstructed (**deserialization**) later in the same or another computer environment" (ref: wikipedia)

- In Java, serialization is usually used to save/read objects to/from files using ObjectOutputStream and ObjectInputStream

- A class must implement **java.io.Serializable** interface to be serializable. It is a marker interface (has no methods to implement)

Serialization Rules in Java

- All primitive types are serializable.
- Transient fields (with transient modifier) are NOT serialized, (i.e., not saved or restored). A class that implements Serializable must mark transient fields of classes that do not support serialization (e.g., a file stream).
- Static fields (with static modifier) are Not serialized.
- If member variables of a serializable object reference to a non-serializable object, the code will compile but a RuntimeException will be thrown.

Example Program: save/read the students

```
public class Student
        implements java.io.Serializable {

    private int id;
    private String firstName;
    private String lastName;
    transient private String dummy;

    public Student(int id, String firstName,
            String lastName, String dummy) {
            this.id = id;
            this.firstName = firstName;
            this.lastName = lastName;
            this.dummy = dummy;
    }
```

```
    // getters and setters are written here

    public String toString() {
            return id + " - " +
                    firstName + " " +
                    lastName +  " " +
                    dummy;
    }

}
```

Program.java

```
public static void main(String[] args) throws Exception {
    // create students
    Student[] students = new Student[2];
    students[0] = new Student(20131234, "Ali", "Doğru", "dummy1");
    students[1] = new Student(20135678, "Veli", "Yanlış", "dummy2");
    // create the file
    ObjectOutputStream oos = new ObjectOutputStream(new
                    FileOutputStream(new File("students.dat")));
    for (int i = 0; i < students.length; i++) {
        oos.writeObject(students[i]); // write the object to file serializing
        System.out.println(students[i]);  // print the object
    }
    oos.close();            // close the file
```

Program.java (continued)

```
    // let's read and display the saved objects on the screen

    // open the file
    ObjectInputStream ois = new ObjectInputStream(
                    new FileInputStream(new File("students.dat")));
    for (int i = 0; i < students.length; i++) {
        // read the student object from file deserializing
        Student s = (Student) ois.readObject();
        System.out.println(s);
    }
    ois.close();
}
```

Output of the program

Objects written to the file:

20131234 - Ali Doğru - **dummy1**

20135678 - Veli Yanlış - **dummy2**

Objects read from the file:

20131234 - Ali Doğru - **null**

20135678 - Veli Yanlış – **null**

Note that, transient field named dummy is not serialized. So, it is null when the objects are deserialized!

java.nio.*

- Be aware of a bit more complex library of Java: The "new" I/O
- It was introduced in JDK 1.4 in the java.nio.* packages
- It’s main goal is speed. It uses ***channels*** and ***buffers*** for I/O (closer to the operating system’s way of performing I/O)
- It supports a non-blocking I/O model.

Summary

- A stream is an object that either
  - Delivers data from your program to a destination, such as a file or screen, (output stream) or
  - Takes data from a source, such as a file or the keyboard, and delivers data to your program (input stream)
- Files are handled as text or binary files
- Java has classes to handle binary (byte oriented) or text (character oriented) files
- Decoration is used to give extra functionality to existing objects. Java I/O library benefits the decoration pattern
- Java supports both sequential and random file access
- Serialization is the job of converting an object to a bit stream that can be saved or transferred to be deserialized later
- Java's nio library is a fast option for I/O

Acknowledgements

- The course material used to prepare this presentation is partially taken/adopted from the list below:
  - Thinking in Java 4th Ed., Bruce Eckel, Prentice Hall, 2006
  - Java - An Introduction to Problem Solving and Programming, Walter Savitch, Pearson, 2012

BBM 102 – Introduction to Programming II

Spring 2017

Introduction to Java

Instructors: *Ayça Tarhan, Fuat Akal, Gönenç Ercan, Vahid Garousi*
TAs: *Selma DİLEK, Selim YILMAZ, Selman BOZKIR*

Today

- **Introduction**
  - About the class
  - Organisation of this course

- **Introduction to Java**
  - Java as a Platform
  - Your First Java Program
  - Basic Programming Elements

Today

- **Introduction**
  - About the class
  - Organisation of this course

- **Introduction to Java**
  - Java as a Platform
  - Your First Java Program
  - Basic Programming Elements

About the course

- This course will help students understand object-oriented programming principles and apply them in the construction of Java programs.
  - The course is structured around basic topics such as *classes, objects, encapsulation, inheritance, polymorphism, abstract classes and interfaces and exception handling*.

- **BBM 104 Introduction to Programming Practicum**: The students will gain hand-on experience via a set of programming assignments supplied as complementary.

- **Requirements**: You must know basic programming (i.e. BBM101).

BBM 102-104 Team

Instructors

Ayça Tarhan
(Section 1)

Fuat Akal
(Section 2)

Gönenç Ercan
(Section 3)

Vahid Garousi
(Section 4)

TAs

Selma DİLEK

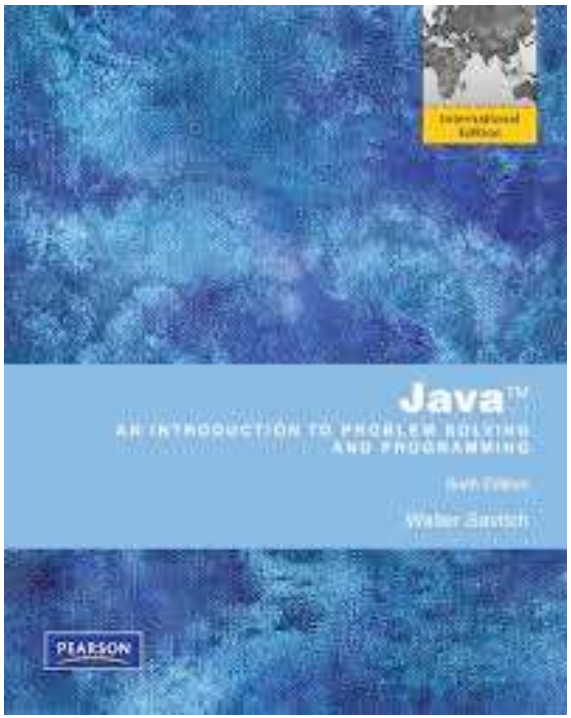
Selim YILMAZ

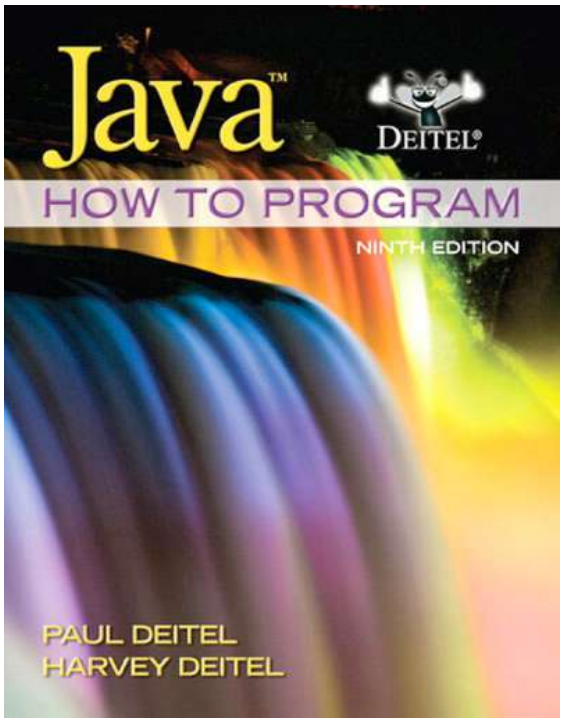
Selman BOZKIR

- **Office hours:** See Web page

Reference Book

- Java - An Introduction to Problem Solving and Programming, Walter Savitch, Pearson, 2012
- Java - How to Program, Paul Deitel and Harvey Deitel, Prentice Hall, 2012





Communication

- The course web page will be updated regularly throughout the semester with lecture notes, programming assignments, announcements and important deadlines.

  http://web.cs.hacettepe.edu.tr/~bbm102

Getting Help

- **Office hours**
  - See the web page for details

- **BBM 104 Introduction to Programming Practicum**
  - Course related recitations, practice with example codes, etc.

- **Communication**
  - Announcements and course related discussions through piazza

BBM 102: https://piazza.com/hacettepe.edu.tr/spring2017/bbm102

BBM 104: https://piazza.com/hacettepe.edu.tr/spring2017/bbm104

Course Work and Grading

- **2 midterm exams (20 + 30 = 50%)**
  - Closed book and notes
  - On April 5th and May 3rd, respectively.

The joy of learning

- **Final exam (45%)**
  - Closed book
  - To be scheduled by the registrar

- **Class Attendance (5%)**
  - Attempting to create false attendance (e.g., signing in the attendance list on behalf of someone else) will be punished.

Course Overview

Week	Date	Title
1	15-Feb	Introduction to Java
2	22-Feb	Introduction to Objects
3	1-Mar	Classes and Objects in Java
4	8-Mar	Encapsulation
5	15-Mar	Inheritance
6	22-Mar	Wrap-up
7	29-Mar	Polymorphism
8	5-Apr	Midterm Exam 1
9	12-Apr	Abstract Classes and Interfaces
10	19-Apr	Collections
11	26-Apr	Exceptions
12	3-May	Midterm Exam 2
13	10-May	Streams and Input/Output
14	17-May	Wrap-up

BBM 104 Introduction to Programming Practicum

Programming assignments (PAs)

- Four assignments throughout the semester.
- Each assignment has a well-defined goal such as solving a specific problem.
- You **must work alone** on all assignments stated unless otherwise.

Important Dates

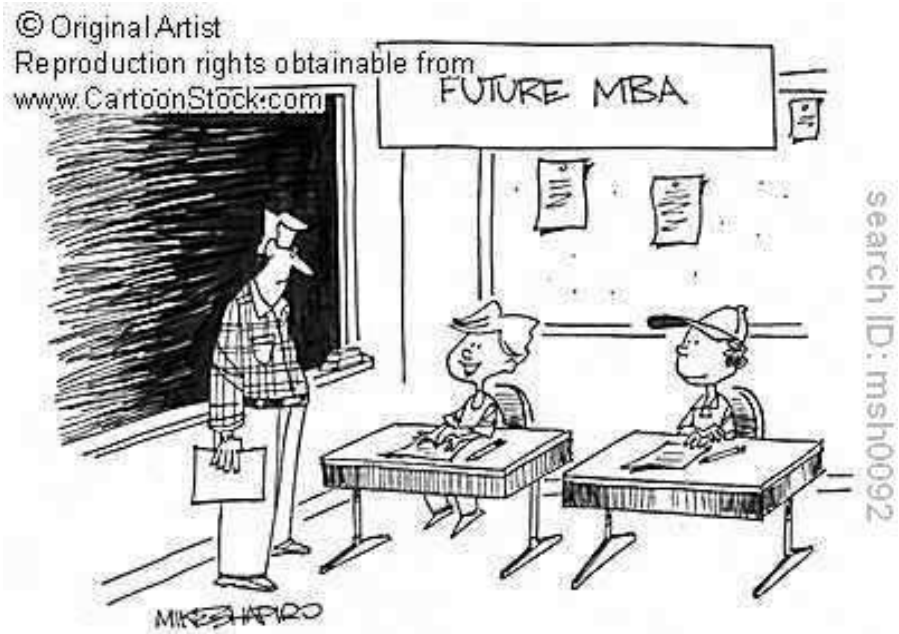
- See the course web page for schedule.

Policies

- **Work groups**
  - You must work alone on all assignments stated unless otherwise
- **Submission**
  - Assignments due at 23:59 (no extensions!)
  - Electronic submissions (no exceptions!)
- **Lateness penalties**
  - Get penalised **10% per day**
  - No late submission is accepted **3 days after due date**

Cheating

- **What is cheating?**
  - Sharing code: by copying, retyping, looking at, or supplying a file
  - Coaching: helping your friend to write a programming assignment, line by line
  - Copying code from previous course or from elsewhere on WWW



"We weren't cheating. We were consulting."

- **What is NOT cheating?**
  - Explaining how to use systems or tools
  - Helping others with high-level design issues

Cheating

- **Penalty for cheating:**
  - Removal from course with failing grade

- **Detection of cheating:**
  - We do check: Our tools for doing this are much better than most cheaters think!

Today

- Introduction
  - About the class
  - Organization of this course

- **Introduction to Java**
  - Java as a Platform
  - Your First Java Program
  - Basic Programming Elements

What is Java?

- *An **island** of Indonesia lying between the Indian Ocean and the Java Sea.*

What is Java?

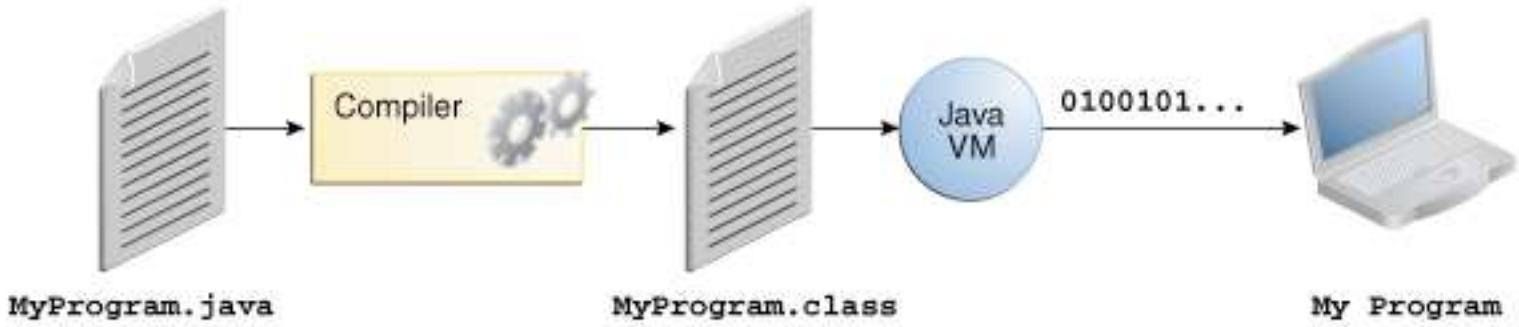
- *Informal. Brewed **coffee**.*

What is Java?

- A technology which is both a programming language and a platform.
- Developed by Sun Microsystems.
- First public version was released in 1995.

Software Development with Java

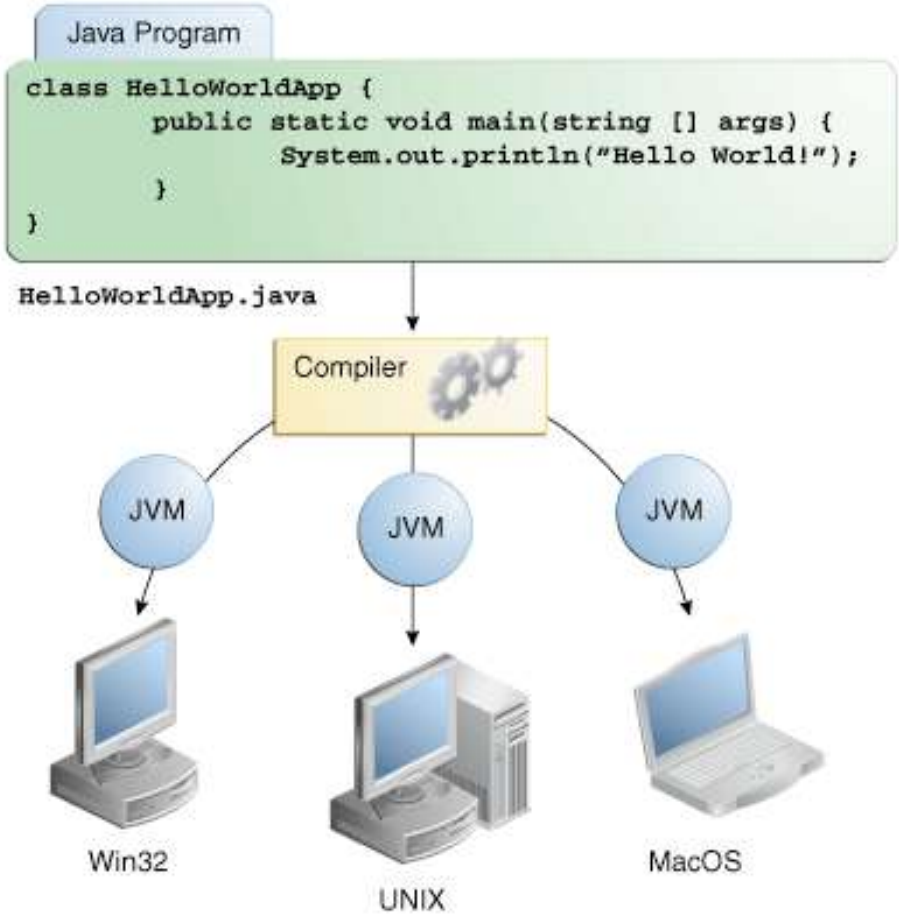
- All source code is first written in plain text files ending with the ".java" extension.
- Those source files are then compiled into ".class" files by the javac compiler.
- A ".class" file does not contain code that is native to your processor; it instead contains *bytecodes* — the machine language of the Java Virtual Machine (Java VM).
- The java launcher tool then runs your application with an instance of the Java Virtual Machine, i.e. your code is run by JVM.



http://docs.oracle.com/javase/tutorial/getStarted/intro/definition.html

Platform Independence: Write Once Run Anywhere

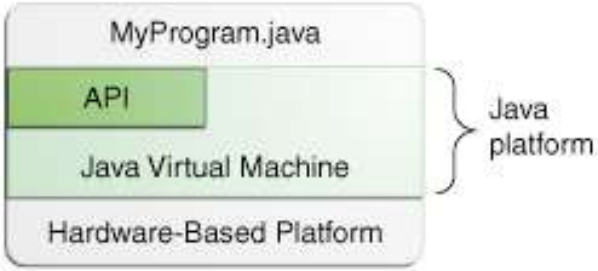
- Because the Java VM is available on many different operating systems, the same .class files are capable of running on Microsoft Windows, the Solaris™ Operating System (Solaris OS), Linux, or Mac OS.



http://docs.oracle.com/javase/tutorial/getStarted/intro/definition.html

The Java Platform

- A *platform* is the hardware or software environment in which a program runs.
- The Java platform has two components:
  - The *Java Virtual Machine:* It's the base for the Java platform and is ported onto various hardware-based platforms
  - The *Java Application Programming Interface* (API): It is a large collection of ready-made software components that provide many useful capabilities.



- As a platform-independent environment, the Java platform can be a bit slower than native code.
  - However, advances in compiler and virtual machine technologies are bringing performance close to that of native code without threatening portability.

http://docs.oracle.com/javase/tutorial/getStarted/intro/definition.html

Your First Java Program

HelloWorld.java

```
public class HelloWorld {

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

}
```

```
$ javac HelloWorld.java ←  Compile
$ java HelloWorld       ←  Run
Hello world!
```

Basic Programming Elements

- Variables, Types and Expressions
- Flow of Control
  - Branching
  - Loops

Variables

- **Variables** in a program are used to store data such as numbers and letters. They can be thought of as containers of a sort.
- You should choose variable names that are helpful. Every variable in a Java program must be declared before it is used for the first time.
- A variable declaration consists of a type name, followed by a list of variable names separated by commas. The declaration ends with a semicolon.

```
Syntax:
data_type variable_name [ = initial_value ];
```

```
int styleNumber, numberOfChecks, numberOfDeposits;
double amount, interestRate;
char answer;
```

Primitive Data Types

Type Name	Kind of Value	Memory Used	Range of Values
byte	Integer	1 byte	-128 to 127
short	Integer	2 bytes	-32,768 to 32,767
int	Integer	4 bytes	-2,147,483,648 to 2,147,483,647
long	Integer	8 bytes	-9,223,372,036,8547,75,808 to 9,223,372,036,854,775,807
float	Floating-point	4 bytes	±3.40282347 × 10+38 to ±1.40239846 × 10–45
double	Floating-point	8 bytes	±1.79769313486231570 × 10+308 to ±4.94065645841246544 × 10–324
char	Single character (Unicode)	2 bytes	All Unicode values from 0 to 65,535
boolean		1 bit	True or false

There are also Class Data Types which we will cover later.

Identifiers

- The technical term for a name in a programming language, such as the name of a variable, is an **identifier.**
- An identifier can contain only letters, digits 0 through 9, and the underscore character "_".
- The first character in an identifier cannot be a digit.
- There is no limit to the length of an identifier.
- Java is **case sensitive** (e.g., *personName* and *personname* are two different variables).

Identifier	Valid?
outputStream	Yes
4you	No
my.work	No
FirstName	Yes
_tmp	Yes
Public	No

Public is a reserved word.

Java Reserved Words

abstract	assert	boolean	break	byte	case
catch	char	class	const	continue	default
double	do	else	enum	extends	FALSE
final	finally	float	for	goto	if
implements	import	instanceof	int	interface	long
native	new	null	package	private	protected
public	return	short	static	strictfp	super
switch	synchronized	this	throw	throws	transient
TRUE	try	void	volatile	while	

Naming Conventions

- Class types begin with an uppercase letter (e.g. **String**).
- Primitive types begin with a lowercase letter (e.g. **float**).
- Variables of both class and primitive types begin with a lowercase letters (e.g. **firstName**, **classAverage**).
- Multiword names are "punctuated" using uppercase letters.

Assignment Statements

- An assignment statement is used to assign a value to a variable.
- The "equal sign" is called the *assignment operator*
- Syntax:

```
variable_name = expression;
```

where **expression** can be another variable, a *literal* or *constant*, or something to be evaluated by using *operators*.

```
amount = 100;
interestRate = 0.12;

answer = 'Y';
fullName = firstName + " " + lastName;
```

Initializing Variables

- A variable that has been declared, but no yet given a value is said to be *uninitialized.*
- Uninitialized class variables have the value **null**.
- Uninitialized primitive variables may have a default value.

Data Type	Default Value
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
String (or any object)	null
boolean	FALSE

- It's good practice **not** to rely on a default value.

Constants

- Literal expressions such as **2, 3.7,** or **'y'** are called *constants*.

- Integer constants can be preceded by a **+** or **-** sign, but cannot contain commas.

- Floating-point constants can be written with digits after a decimal point or using *e notation.*
  - `765000000.0` can be written as `7.65e8`
  - `0.000483` can be written as `4.83e-4`

Imprecision in Floating Point Numbers

- Floating-point numbers often are only approximations since they are stored with a finite number of bits.
- Hence **1.0/3.0** is slightly less than 1/3.
- **1.0/3.0 + 1.0/3.0 + 1.0/3.0** is less than 1.

Named Constants

- Java provides a mechanism that allows you to define a variable, initialise it, and moreover fix the variable's value so that it cannot be changed.

```
public static final Type Variable = Constant;
```

- The convention for naming constants is to use all uppercase letters, with an underscore symbol "_" between words.

```
public static final double PI = 3.14159;
public static final int DAYS_PER_WEEK = 7;
…
float area = PI * r * r ;
int daysInYear = 52 * DAYS_PER_WEEK ;
```

Assignment Compatibility

- Java is *strongly typed.*

- A value of one type can be assigned to a variable of any type further to the right (not to the left):

  byte → short → int → long → float → double

- You can assign a value of type **char** to a variable of type **int**.

Type Conversion (Casting)

- Implicit conversion

```
double doubleVariable = 5;              // 5.0
int intVariable = 5;                    // 5
doubleVariable = intVariable;           // 5.0
```

- Explicit conversion

```
double doubleVariable = 5.0;
int intVariable = doubleVariable ;            // Illegal
int intVariable = (int) doubleVariable ;      // Legal, 5
```

Operators and Precedence

- Precedence
  - First: The unary operators: plus (+), minus(-), not (!), increment (++) and decrement (--)
  - Second: The binary arithmetic operators: multiplication (*), integer division (/) and modulus (%)
  - Third: The binary arithmetic operators: addition (+) and subtraction (-)
- When binary operators have equal precedence, the operator on the left acts before the operator(s) on the right.
- When unary operators have equal precedence, the operator on the right acts before the operation(s) on the left.
- Parenthesis can change the precedence.

Operators and Precedence - Example

Ordinary Math	Java (Preferred Form)	Java (Parenthesized)
$rate^2 + delta$	`rate * rate + delta`	`(rate * rate) + delta`
$2(salary + bonus)$	`2 * (salary + bonus)`	`2 * (salary + bonus)`
$\frac{1}{time + 3mass}$	`1 / (time + 3 * mass)`	`1 / (time + (3 * mass))`
$\frac{a-7}{t+9v}$	`(a - 7) / (t + 9 * v)`	`(a - 7) / (t + (9 * v))`

Figure from "Java - An Introduction to Problem Solving and Programming, Walter Savitch, Pearson, 2012"

Specialised Assignment Operators

- You can precede the simple assignment operator (=) with an arithmetic operator (+, -, *, /, %) to produce a kind of special-purpose assignment operator.

`amount += 5;`	equals to	`amount = amount + 5;`
`amount *= 5;`	equals to	`amount = amount * 5;`

Increment / Decrement Operators

- Used to increase (or decrease) the value of a variable by 1

- The increment operator
  - count++ → Use the value of count and then increase it.
  - ++count → Increase the value of count and then use it.

- The decrement operator
  - count-- → Use the value of count and then decrease it.
  - --count → Decrease the value of count and then use it.

Increment / Decrement Operators - Example

- The increment operator (prefix form)

```
int n = 3;
int m = 4;
int result = n * (++m);  // result = 15
```

- The increment operator (postfix form)

```
int n = 3;
int m = 4;
int result = n * (m++); // result = 12
```

Arrays

- Array is a sequence of values.
- Array indices begin at zero.
- Defining Arrays

```
Base_Type[] Array_Name = new Base_Type[Length];
```

```
int[] numbers = new int[100];            // or,
```

```
int[] numbers;
numbers = new int[100];
```

- Initialising Arrays

```
double[] reading = {3.3, 15.8, 9.7};     // or,
```

```
double[] reading = new double[3];
reading[0] = 3.3;
reading[1] = 15.8;
reading[2] = 9.7;
```

Strings

- A value of type **String** is a
  - Sequence (Array) of characters treated as a single item
  - Character positions start with 0

Indices	0	1	2	3	4	5	6	7	8	9	10	11
	J	a	v	a		i	s		f	u	n	.

Note that the blanks and the period count as characters in the string.

- Can be declared in three ways:

```
String greeting;
greeting = "Hello World!”;
```

```
String greeting = "Hello World!”;
```

```
String greeting = new String("Hello World!”);
```

Figure from “Java - An Introduction to Problem Solving and Programming, Walter Savitch, Pearson, 2012”

Concatenating Strings

- You can connect—or join or paste—two strings together to obtain a larger string. This operation is called **concatenation** and is performed by using the “+” operator.

```
String greeting, sentence;
greeting = "Hello”;

sentence = greeting + “ my friend!”;
System.out.println(sentence);     // Hello my friend!
```

```
String solution = “The answer is ” + 42;
System.out.println(solution);     // The answer is 42

// Java converts the number constant 42 to the
// string constant "42" and then concatenates the
// two strings
```

String Methods

- **Homework**: Investigate the methods given below. You will be responsible in the exams.

charAt (Index)	length()
compareTo(A_String)	replace(OldChar, NewChar)
concat(A_String)	substring(Start)
equals(Other_String)	substring(Start,End)
equalsIgnoreCase(Other_String)	toLowerCase()
indexOf(A_String)	toUpperCase()
lastIndexOf(A_String)	trim()

Boolean Type

- Java has the logical type `boolean`
- Type `boolean` has two literal constants
  - `true`
  - `false`

```
int number = -5;
boolean isPositive = (number > 0);        // False
```

Java Comparison Operators

Math Notation	Name	Java Notation	Java Examples
=	Equal to	`==`	`balance == 0` `answer == 'y'`
≠	Not equal to	`!=`	`income != tax` `answer != 'y'`
>	Greater than	`>`	`expenses > income`
≥	Greater than or equal to	`>=`	`points >= 60`
<	Less than	`<`	`pressure < max`
≤	Less than or equal to	`<=`	`expenses <= income`

Figure from "Java - An Introduction to Problem Solving and Programming, Walter Savitch, Pearson, 2012"

Java Logical Operators

Name	Java Notation	Java Examples
Logical *and*	&&	`(sum > min) && (sum < max)`
Logical *or*	||	`(answer == 'y') || (answer == 'Y')`
Logical *not*	!	`!(number < 0)`

Figure from "Java - An Introduction to Problem Solving and Programming, Walter Savitch, Pearson, 2012"

Flow of Control

- *Flow of control* is the order in which a program performs actions.
- A *branching statement* chooses between two or more possible actions.
  - If-else, switch statements
- A *loop statement* repeats an action until a stopping condition occurs.
  - For, while, do-while loops

Basic if Statement

- Syntax

```
if (Expression)
    Action
```

- If the *Expression* is true then execute *Action*
- *Action* is either a single statement or a group of statements within braces

```
if (value2 < value1) {      // Rearrange numbers so
        int tmp = value1;   // value2 variable should
        value1 = value2;    // hold the bigger value
        value2 = tmp;
}
```

if-else Statement

- Syntax

```
if (Expression)
    Action1
else
    Action2
```

- If *Expression* is true then execute *Action1* otherwise execute *Action2*
- The actions are either a single statement or a list of statements within braces

```
int maximum;
if (value1 < value2) {    // is value2 larger?
   maximum = value2;      // yes: value2 is larger
}
else {                    // (value1 >= value2)
   maximum = value1;      // no: value2 is not larger
}
```

if-else-if Statement

- If statements can be nested (also called as multi-way, multi-branch if statement)

```
if (a == '0')
      System.out.println ("zero");
else if (a == '1')
      System.out.println ("one");
else if (a == '2')
      System.out.println ("two");
else if (a == '3')
      System.out.println ("three");
else if (a == '4')
      System.out.println ("four");
else
      System.out.println ("five+");
```

Switch Statement

- Switch statement can be used instead of multi-way if statement.
- Syntax

```
switch(controlling_expression)  {
        case expression1:
                action1;
                break;
        case expression2:
                action2;
                break;
        …
        default:
                actionN;
}
```

- Every case ends with *break* statement.

Switch Statement

- Switch statements are more readable than nested if statements

```
switch (a) {
    case '0':
        System.out.println ("zero"); break;
    case '1':
        System.out.println ("one"); break;
    case '2':
        System.out.println ("two"); break;
    case '3':
        System.out.println ("three"); break;
    case '4':
        System.out.println ("four"); break;
    default:
        System.out.println ("five+"); break;
}
```

The Conditional (Ternary) Operator

- The `?` and `:` together are called the *conditional operator* or *ternary operator.*

```
if (n1 > n2)
   max = n1;
else
   max = n2;
```

can be written as:

```
max = (n1 > n2) ? n1 : n2;
```

for Loops

- The for loop is a pretest loop statement. It has the following form.

```
for (initialisation; boolean-expression; increment){
    nested-statements
}
```

- *initialisation* is evaluated first.
- *boolean-expression* is tested *before* each iteration of the loop.
- *increment* is evaluated at the end of each iteration.
- *nested-statements* is a sequence of statements. If there is only one statement then the braces may be omitted

Varying Control Variable

- for (int i = 1; i <= 100; i++)
  - from 1 to 100 in increments of 1
- for (int i = 100; i >= 1; i--)
  - from 100 to 1 in increments of -1
- for (int i = 7; i <= 77; i += 7)
  - from 7 to 77 in increments of 7
- for (int i = 20; i >= 2; i -= 2)
  - from 20 to 2 in decrements of 2

For Loop Example

```
String[] classList = {"Jean", "Claude", "Van",
                      "Damme"};

for (int i=0; i<classList.length; i++) {
      System.out.println(classList[i]);
}
```

```
Jean
Claude
Van
Damme
```

```
for (String name : classList) {
      System.out.println(name);
}
```

```
Jean
Claude
Van
Damme
```

While Loop

- The while loop is a pretest loop statement. It has the following form.

```
while (boolean-expression) {
      nested-statements
}
```

- *boolean-expression* is an expression that can be true or false.
- *nested-statements* is a sequence of statements. If there is only one statement then the braces can be omitted.
- The boolean expression is tested *before* each iteration of the loop. The loop terminates when it is false.

While Loop Example

```
int[] numbers = { 1, 5, 3, 4, 2 };
int i=0, key = 33;

boolean found = false;

while (!found){
      if (numbers[i++] == key)
            found=true;
}

if (found)
      System.out.println("Key is found in the array");
else
      System.out.println("Key is NOT found!");
```

Let's look for something that does not exist.

Is there a problem here?

While Loop Example

```
int[] numbers = { 1, 5, 3, 4, 2 };
int i=0, key = 33;

boolean found = false;

while (!found && i<numbers.length){
      if (numbers[i++] == key)
            found=true;
}

if (found)
      System.out.println("Key is found in the array");
else
      System.out.println("Key is NOT found!");
```

Make sure that the loop ends somehow.

Figure from "Java - An Introduction to Problem Solving and Programming, Walter Savitch, Pearson, 2012"

Do-While Loop

- The do-while loop is a post-test loop statement. It has the following form.

```
do {
      nested-statements
} while (boolean-expression);
```

- *nested-statements* is a sequence of statements. If there is only one statement then the braces may be omitted.
- *boolean-expression* is an expression that can be true or false.
- The boolean expression is tested *after* each iteration of the loop. The loop terminates when it is false.

Do-While Example

```
Scanner scan = new Scanner(System.in);
int myNumber;

do {
      System.out.println(
            "Enter a number between 0 and 100: ");

      myNumber = scan.nextInt();

} while (!(myNumber >= 0 && myNumber <= 100));

System.out.println("You entered a valid number");
```

Break Statement

- The break statement is used in loop (for, while, and do-while) statements and switch statements to terminate execution of the statement. A break statement has the following form.

```
break;
```

- After a break statement is executed, execution proceeds to the statement that follows the enclosing loop or switch statement.

- Use `break` statements sparingly (if ever).

Continue Statement

- A **continue** statement
  - Ends current loop iteration
  - Begins the next one

- Use of continue statement is not recommended
  - Introduce unneeded complications

Breaking a Loop

```
int[] numbers = { 1, 5, 3, 4, 2 };
int i = 0, key = 3;

while (i < numbers.length) {
      if (numbers[i] == key)
            break;
      i++;
}

if (i < numbers.length)
      System.out.println("Key is found in the array");
else
      System.out.println("Key is NOT!");
```

Summary

- So far, it should be fairly easy to follow for those who has basic programming skills / who has taken BBM101.
- We will continue with objects next week.
- In the mean time, here is a good starting point to Java:

  http://docs.oracle.com/javase/tutorial/index.html
- Also check out these notes by Oğuz Aslantürk in Turkish:

  http://web.cs.hacettepe.edu.tr/~bbm102/misc/java_notes_by_oa.pdf

Acknowledgments

- The course material used to prepare this presentation is mostly taken/adopted from the list below:
  - Java - An Introduction to Problem Solving and Programming, Walter Savitch, Pearson, 2012.
  - Java tutorials http://docs.oracle.com/javase/tutorial/
  - Aaron Bloomfield, CS101, University of Virginia.

BBM 102 – Introduction to Programming II

Spring 2017

Introduction to Object Orientation

Instructors: Ayça Tarhan, Fuat Akal, Gönenç Ercan, Vahid Garousi
TAs: Selma Dilek, Selim Yılmaz, Selman Bozkır

Today

- **Software as a Complex Thing**
  - Dealing with Complexity
  - Functional Decomposition
  - Structured Programming

- **Object Oriented Paradigm**
  - Principles of Object Orientation
  - Classes and Objects
  - Sample Object Designs

Today

- **Software as a Complex Thing**
  - Dealing with Complexity
  - Functional Decomposition
  - Structured Programming

- **Object Oriented Paradigm**
  - Principles of Object Orientation
  - Classes and Objects
  - Sample Object Designs

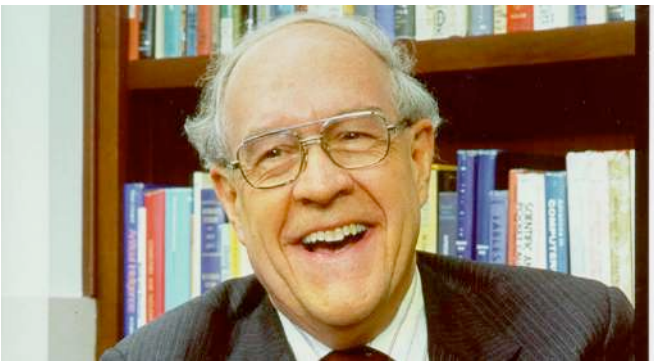
Software in Modern World

- We can't run the modern world without software.
  - National infrastructures and utilities are controlled by computer-based systems
  - Most electrical products include a computer and controlling software
  - Industrial manufacturing and distribution is completely computerized, as is the financial system.
  - Entertainment, including the music industry, computer games, and film and television, is software intensive

Software in Modern World: Examples

Software is a Complex Thing

- Software development is a serious business.
- We are not talking about a program that finds *N Fibonacci* numbers anymore.
- We are talking about *industrial-strength (professional)* software.
  - It is intensely difficult, if not impossible, for the individual developer to comprehend all the subtleties of its design.
- "The complexity of software is an essential property, not an accidental one." -- F.P. Brooks



Professional Software Development

- Professional software, intended for use by someone apart from its developer, is usually *developed by teams* rather than individuals.
  - It is maintained and changed throughout its life.

- Software development is referred to mean *professional software development*, rather than individual programming.
  - It includes techniques that support program specification, design, and evolution, none of which are normally relevant for personal software development.

Professional Software Development

- Many people think that software is simply another word for computer programs.

- However, software is *not just the programs themselves* but also *all associated documentation and configuration data* that is required to make these programs operate correctly.

- This is one of the important differences between professional and amateur software development.

Quality of Software

- When we talk about the quality of professional software, we have to take into account that the software is used and changed by people apart from its developers.
  - Quality is therefore not just concerned with *what the software does*; it has to include the *software's behavior* while it is executing and *the structure and organization of the system programs and associated documentation*.

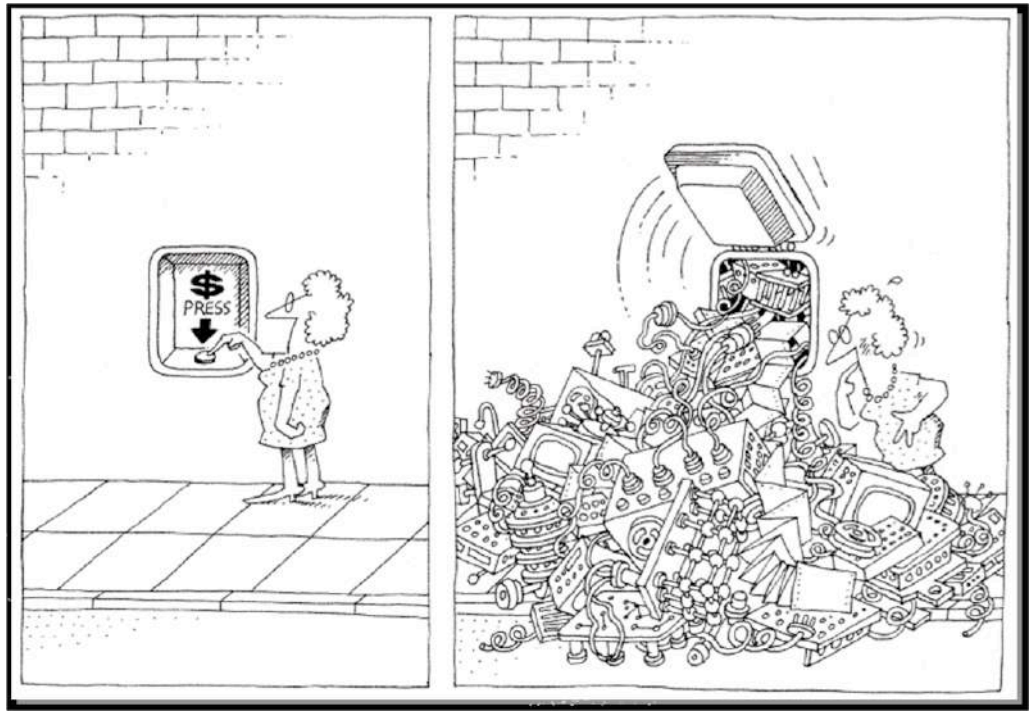
- This is so-called *quality* or *non-functional software attributes*.
  - Examples of these attributes are the *software's response time to a user query* and the *understandability of the program code*.
  - Depends on its application domain. Therefore, a banking system must be secure, an interactive game must be responsive, a telephone switching system must be reliable, and so on.

Essential Attributes of Quality Software

Product characteristics	Description
Maintainability	Software should be written in such a way so that it can evolve to meet the changing needs of customers. This is a critical attribute because software change is an inevitable requirement of a changing business environment.
Dependability and security	Software dependability includes a range of characteristics including reliability, security, and safety. Dependable software should not cause physical or economic damage in the event of system failure. Malicious users should not be able to access or damage the system.
Efficiency	Software should not make wasteful use of system resources such as memory and processor cycles. Efficiency therefore includes responsiveness, processing time, memory utilization, etc.
Acceptability	Software must be acceptable to the type of users for which it is designed. This means that it must be understandable, usable, and compatible with other systems that they use.

Reference: I. Sommerville, Software Enginering (9th ed), Pearson, 2011.

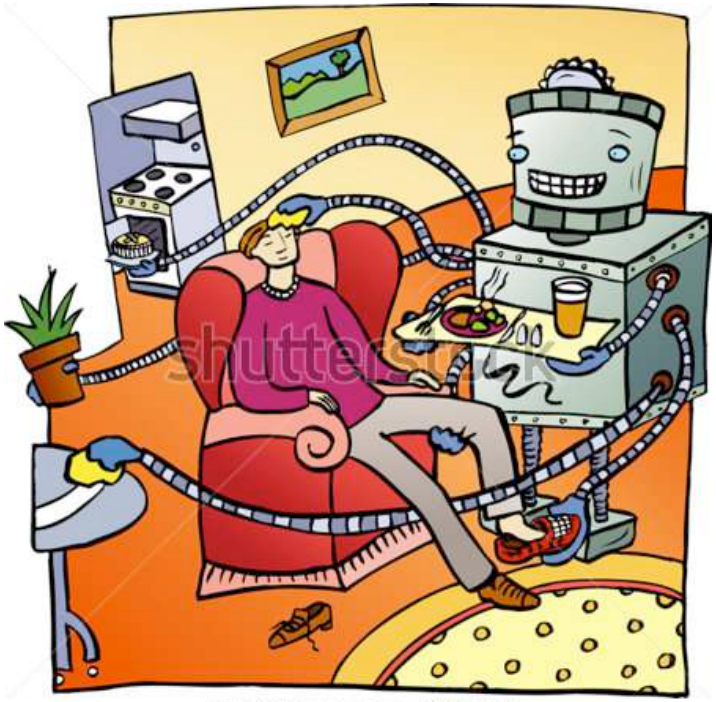
Inherent Complexity of Software



- The complexity of the problem domain
- The difficulty of managing the development process
- The flexibility possible through software
- The problem of characterizing the behavior of discrete systems

The Complexity of the Problem Domain

- The problem often involves elements of *inescapable complexity*, in which we find a myriad of competing, perhaps even contradictory**, requirements**.
  - Consider the requirements for an autonomous robot. Its raw functionality is difficult enough to comprehend.
  - Also consider **nonfunctional requirements** such as usability, performance, survivability, and reliability.

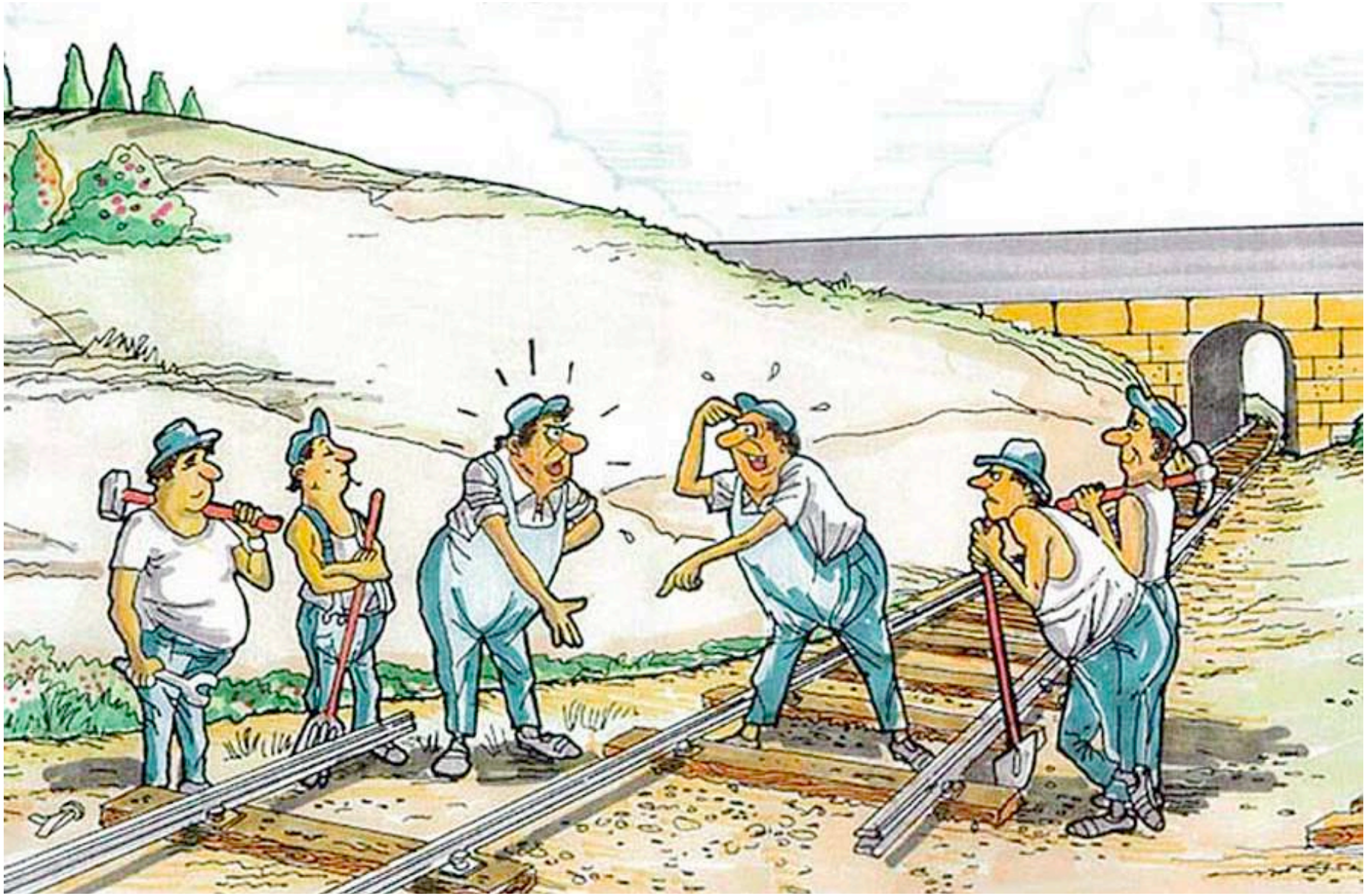


- The requirements of a software system often change during its development.

The Difficulty of Managing the Development Process

- Industrial-strength software typically consists of hundreds and sometimes thousands of **separate modules**.
- No one person can ever understand such a system completely.
- This amount of work demands that we use a **team of developers**, and ideally we use as small a team as possible.
- There are challenges associated with team development.
  - Having more developers means **more complex communication** and more **difficult coordination**, particularly if the team is geographically dispersed.
  - With a team of developers, the key management challenge is always to maintain a unity and integrity of design.

The Difficulty of Managing the Development Process



Software (Development) Process

The systematic approach to software development is called a software process which is defind as a sequence of activities that leads to the production of a software product:

- Software specification, where customers and engineers define the software that is to be produced and the constraints on its operation.
- Software development, where the software is designed and programmed.
- Software validation, where the software is checked to ensure that it is what the customer requires.
- Software evolution, where the software is modified to reflect changing customer and market requirements.

Software (Development) Processes

- Different types of systems need different development processes.
  - For example, *real-time software* in an aircraft has to be completely specified before development begins.
  - In *e-commerce systems*, the specification and the program are usually developed together.

- Consequently, these generic activities may be organized in different ways and described at different levels of detail depending on the type of software being developed.
  - There are many different types of software. There is no universal software engineering method or technique that is applicable for all of these.

The Flexibility Possible through Software

- Software is *abstract* and *changeable*.

- A home-building company generally does not operate its own tree farm from which to harvest trees for construction. In the software industry such practice is common.
  - Software offers the ultimate flexibility, so it is possible for a developer to express almost any kind of **abstraction**.
  - This flexibility forces the developer to craft virtually all the primitive building blocks on which these higher-level abstractions stand.
  - While the construction industry has uniform building codes and standards for the quality of raw materials, few such standards exist in the software industry due to *abstactness of the software*.

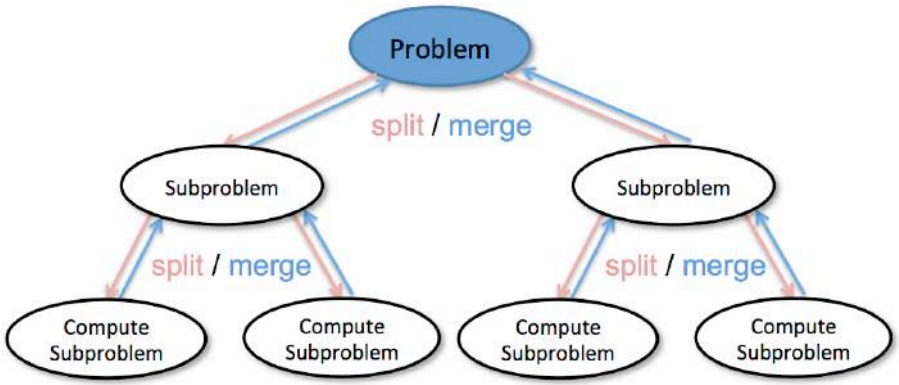
Characterizing the Behavior of Discrete Systems - Difficulties

- There may be hundreds or thousands of variables as well as more than one thread of control.
  - The entire collection of these variables, their current values, and the current address and calling stack of each process within the system constitute the **present state** of the application.

- Each event external to a software system has the potential of placing that system in a **new state**
  - The mapping from state to state is not always deterministic.
  - The event may **corrupt the state of a system** because its designers failed to take into account certain interactions among events.

- **Vigorous testing** is essential but not possible to model the complete behavior of large discrete systems.
  - We must survive with acceptable levels of confidence.

How to Deal with Complexity?



The technique of mastering complexity has been known since ancient times: *divide et impera* (divide and rule)

- When designing a complex software system, it is essential to decompose it into smaller and smaller parts, each of which we may then refine independently.

- In this manner, we satisfy the very real constraint that exists on the channel capacity of human cognition: ***To understand any given level of a system, we need only comprehend a few parts (rather than all parts) at once.***

How Did You Decompose So Far?

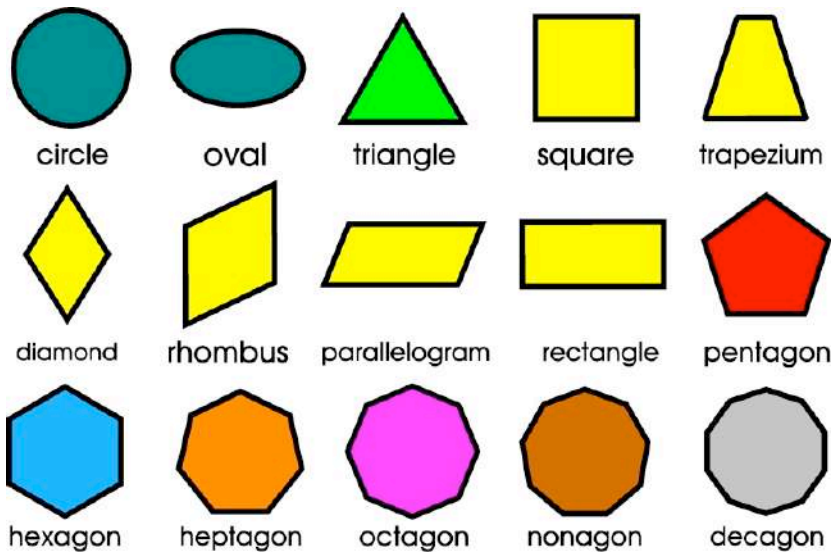
- Functional Decomposition
  - A natural way to deal with complexity
  - Break down (decompose) the problem into the **functional steps** that compose it.

- Example: Write a code to
  - Access a description of shapes that were stored in a database
  - Display those shapes

→*How do you decompose these tasks?*

Functional Decomposition - Example

- It would be natural to think in terms of the steps required:
  1. Locate the list of shapes in the database.
  2. Open up the list of shapes.
  3. Sort the list according to some rules.
  4. Display the individual shapes on the monitor.



Functional Decomposition - Example (cont'd)

- Further breaking down of Step 4 is possible:
  1. Locate the list of shapes in the database.
  2. Open up the list of shapes.
  3. Sort the list according to some rules.
  4. Display the individual shapes on the monitor.
     a) Identify the type of shape.
     b) Get the location of shape.
     c) Call the appropriate function that will display the shape by giving the shape's location.

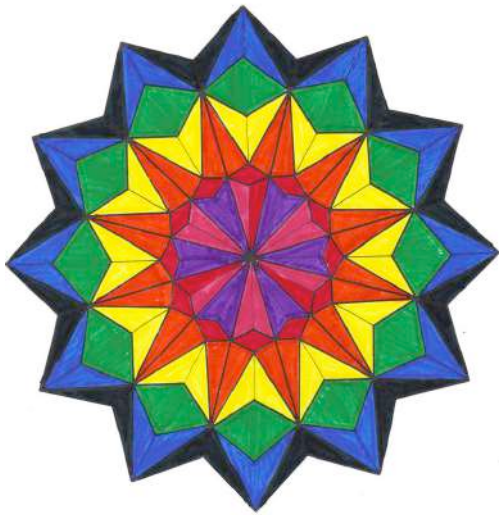
We will revisit this algorithm (Step 4.c more precisely) few slides later.

Is Functional Decomposition Perfect?

- It does not help us much prepare the code for possible changes in the future.

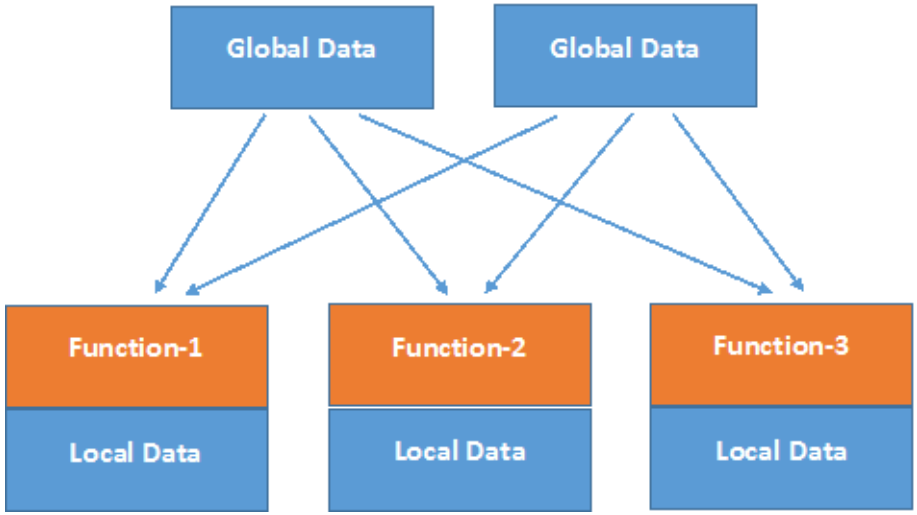
- Many bugs originate with changes to the code.
  - Change creates opportunities for mistakes and unintended consequences.

- Nothing you can do will stop change.
  - You can never get all of requirements from the user.
  - When the business domain changes, so the software.
  - Future is unknown – things will change.

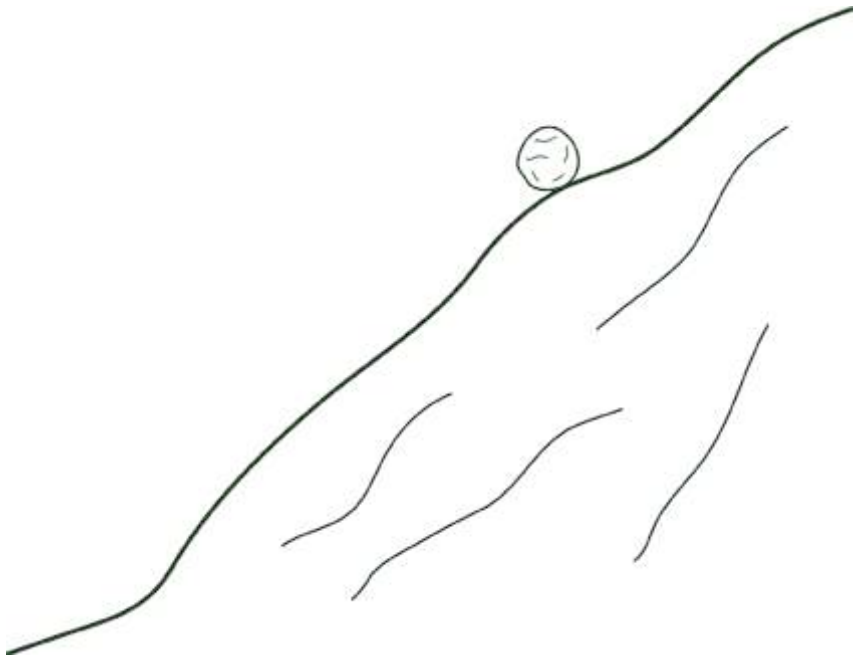


Unwanted Side Effect

- Make a change to a function or a piece of data in one area of the code.
  - Then, have an unexpected impact on other pieces of code.



- Wrong focus: Changes to one set of functions or data impact other sets of functions and other sets of data.
  - Like a snowball that picks up snow as it rolls downhill!



Problem with Requirements

Problem with Requirements

- **Requirements** from users are
  - incomplete, usually wrong, misleading, not telling the whole story.

- **Requirements change** for a very simple set of reasons:
  - The users see new possibilities for the software after discussions with developers.
  - Developers become more familiar with users' problem domain.
  - The environment in which the software is being developed changes.

- You must write your code to accommodate change
  - Not give up on **gathering good requirements**

Let's Take a Close Look on Step 4.c

- Step 4.c of displaying shapes
  - Call appropriate function that will display shape

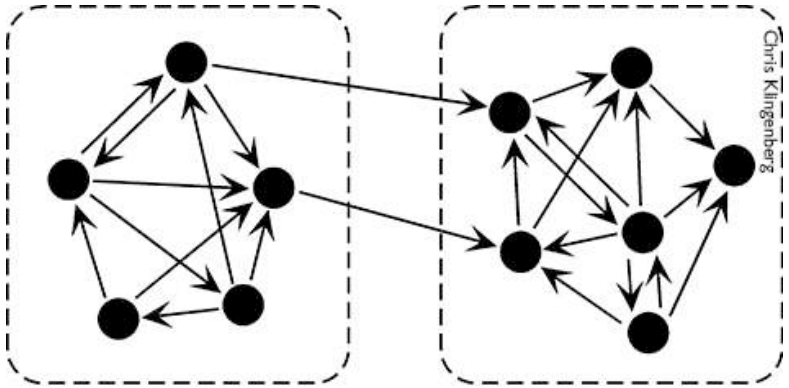
```
function: display shape
input: type of shape, description of shape
action: switch (type of shape)
                case square:
                        put display function for square here
                case circle:
                        put display function for circle here
```

- Use **modularity** to contain variation

Modularity

- May or may not be possible to have a consistent description of shapes that will work for all shapes

- Modularity
  - Makes the code more understandable
  - Understandability makes the code easier to maintain
  - Does not deal with all of the variation it might encounter



Module 1 Module 2

- The goal is to create routines with
  - internal integrity (*strong cohesion* - how closely the operations in a routine are related) and
  - small, direct, visible, and flexible relations to other routines (*loose coupling* -connection between routines)

Structured Programming Approach Example

- Students in your class are having another class. They are going to attend to another class, but they do not know where their next class is located.

- Algorithm:
  1. Get a list of students in the class.
  2. For each student on this list:
     a) Find the next class she/he is taking
     b) Find the location of that class
     c) Find the way to get from your classroom to the student's next class
     d) Tell the student how to get to their next class

- Would you actually follow this approach?
  - Post directions in the back of the room
  - Expect everyone would know what their next class was

What is the Difference?

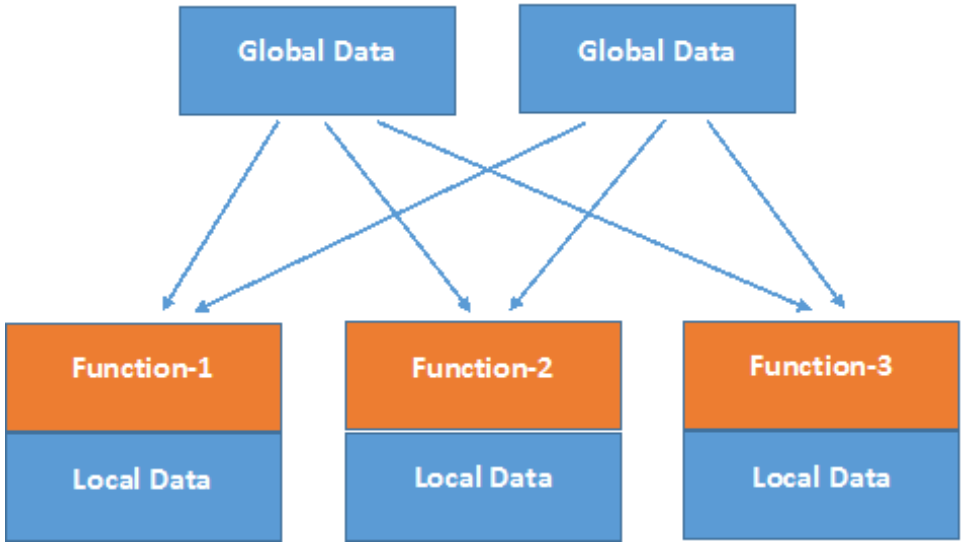
- First case
  - Give explicit directions to everyone.
  - No one other than you is responsible for anything.

- Second case
  - Give general instructions
  - Each person figure out how to do the task himself or herself

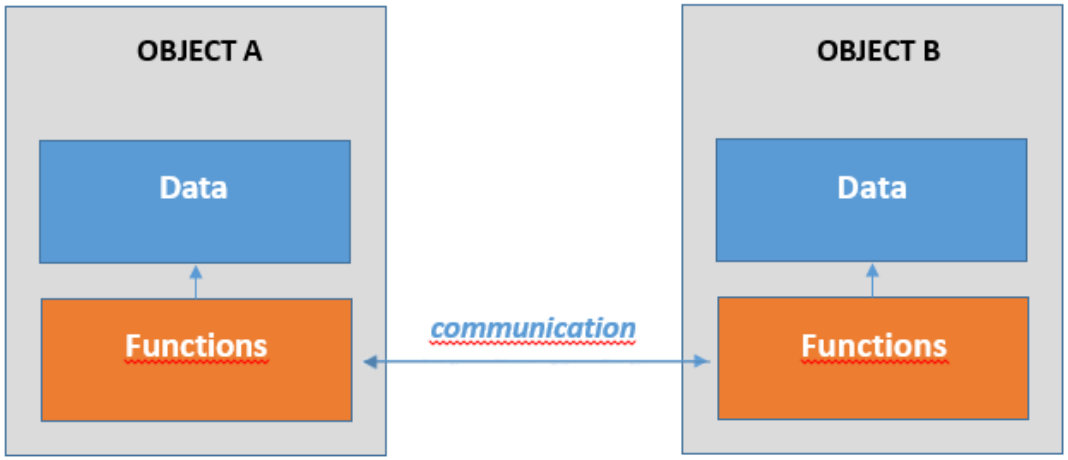
- Shift of responsibility may not be a bad thing

Srtuctured versus Object-Oriented Programming Constructs

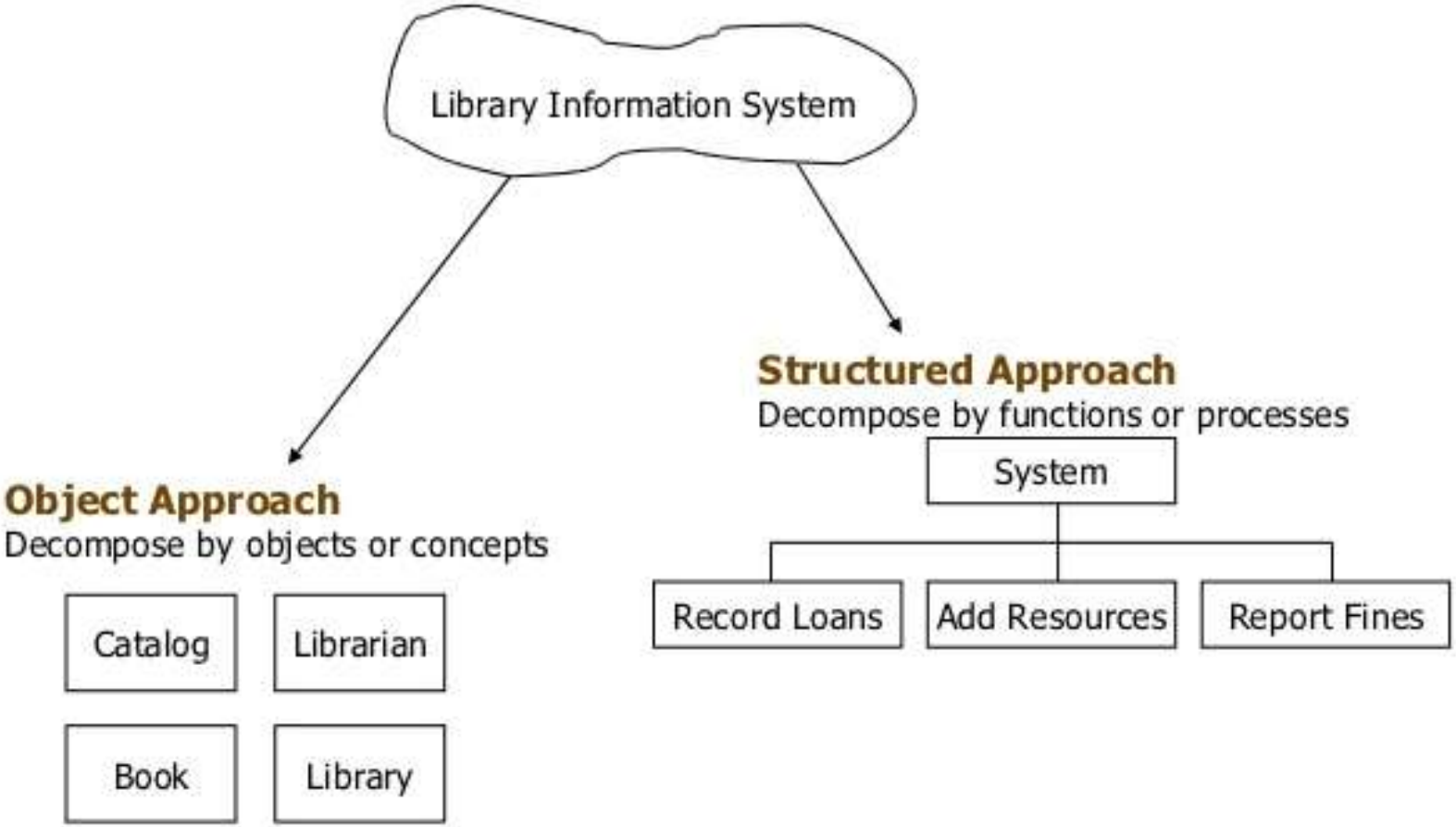
Structured



Object-oriented



Modularity: Structured vs. Object-oriented



Source: Craig Larman, *Applying UML and Patterns* (Prentice Hall, 1998), p. 14

Today

- **Software as a Complex Thing**
  - Dealing with Complexity
  - Functional Decomposition
  - Structured Programming

- **Object Oriented Paradigm**
  - Principles of Object Orientation
  - Classes and Objects
  - Sample Object Designs

"Sir Isaac Newton secretly admitted to some friends: He understood how gravity behaved, but not how it worked!"

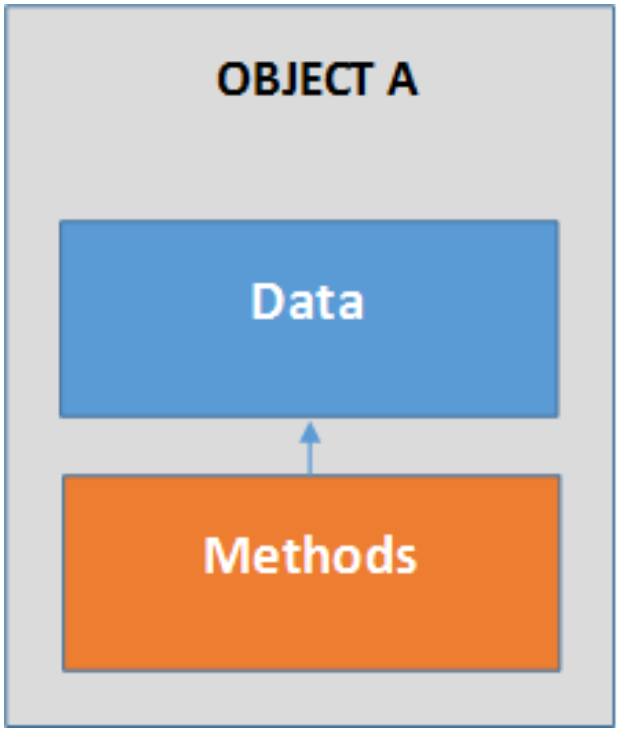
LILY TOMLIN

The Search for Signs of Intelligent Life in the Universe

Object-Oriented Paradigm

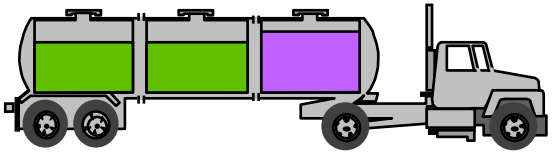
- Centered on the concept of the object

- Object
  - Is data with methods
    - Data (**attributes**) can be simple things like number or character strings, or they can be other objects.
  - Defines things that are responsible for themselves
    - Data to know what state the object is in.
    - **Method** (code) to function properly.



What is an Object?

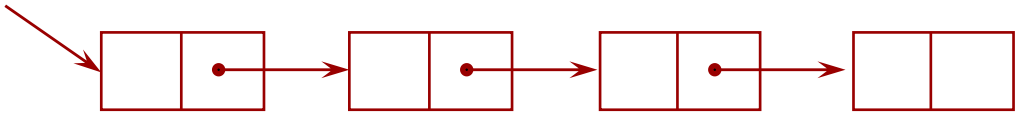
- Informally, an object represents an entity which is either physical, conceptual or software.
  - Physical entity
  - Conceptual entity
  - Software entity



Truck

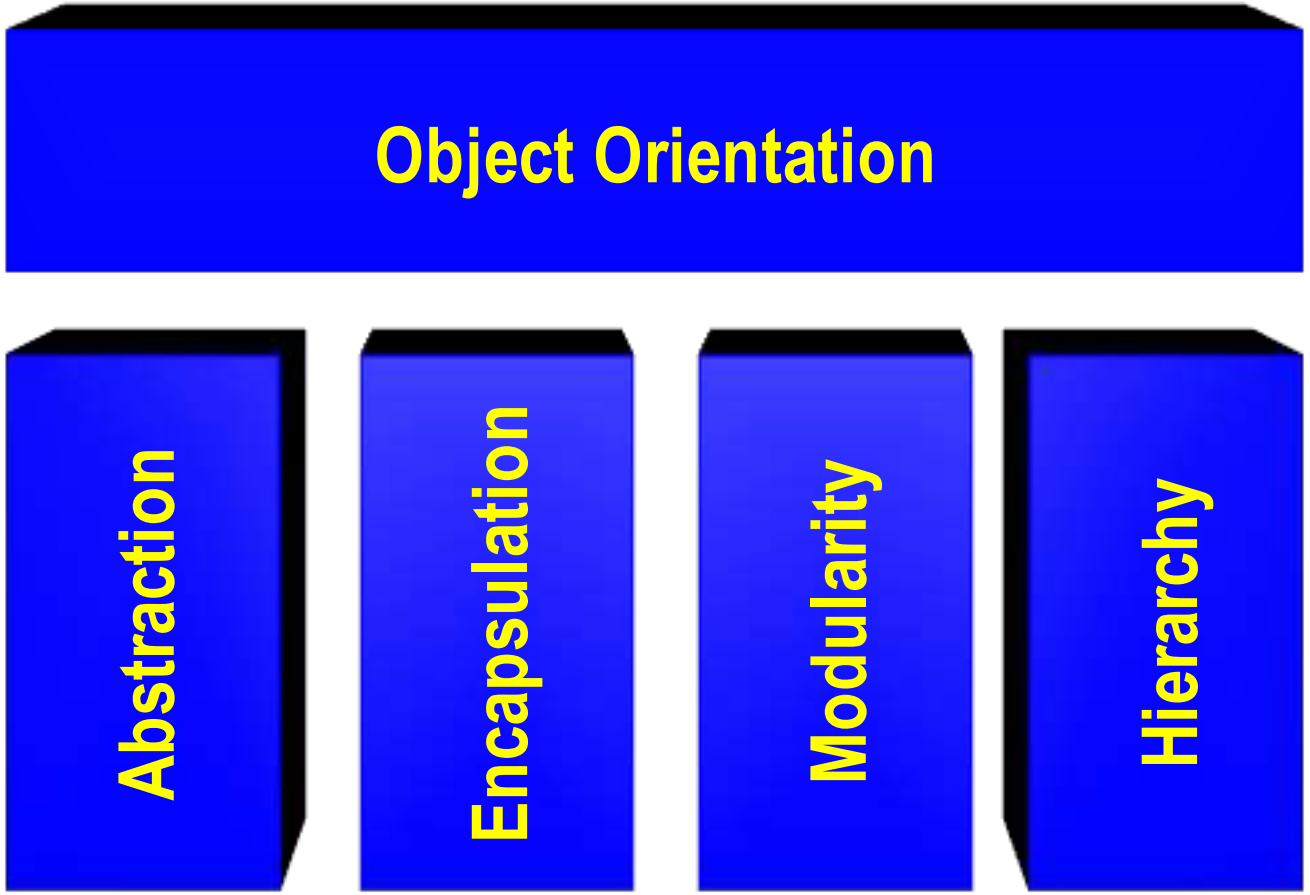
Chemical Process

Bank Account



Linked List

Basic Principles of Object Orientation



What is Abstraction?

Abstraction is one of the fundamental ways that we as humans cope with complexity.

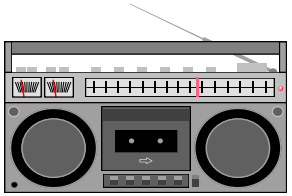
Salesperson

Customer

Not saying

which salesperson

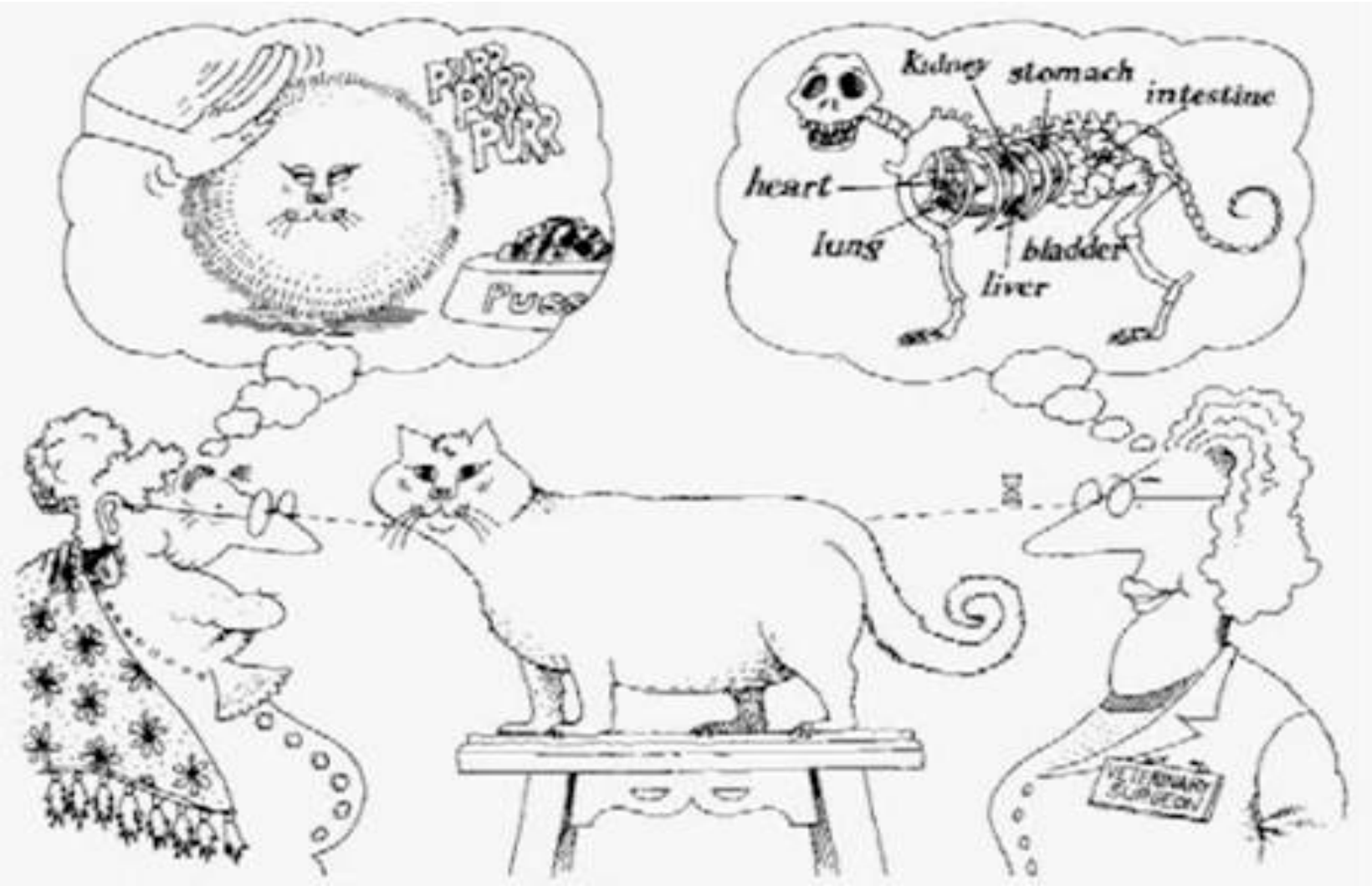
just a salesperson in general!



Product

Dahl, Dijkstra, and Hoare suggest that "abstraction arises from a **recognition of similarities** between certain objects, situations, or processes in the real world, and the decision to ***concentrate upon these similarities and to ignore for the time being the differences***".

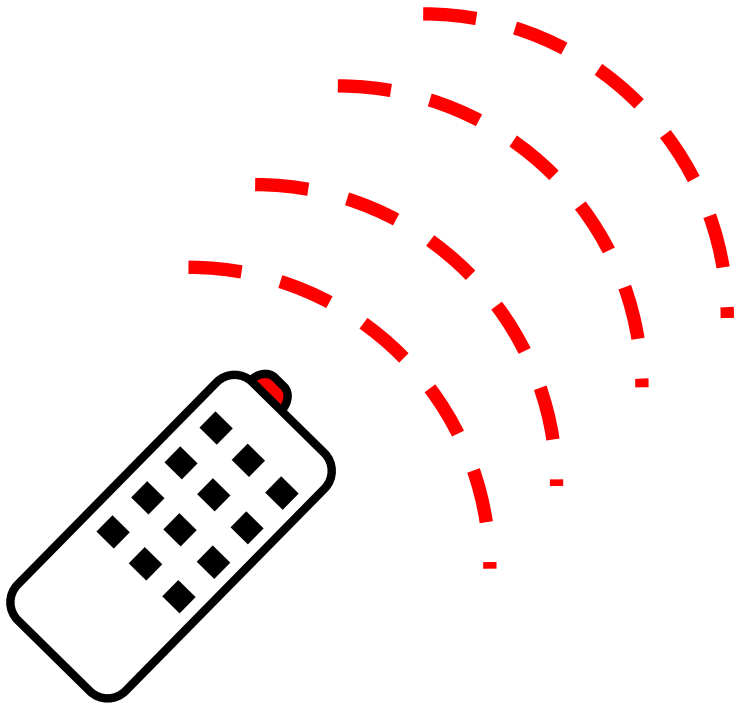
What is Abstraction?

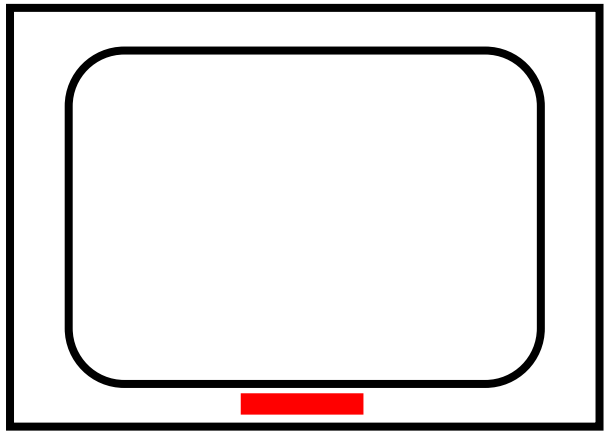


Abstraction focuses upon the essential characteristics of some object, relative to the perspective of the viewer.

What is Encapsulation?

- Hide implementation from clients
  - Clients depend on interface

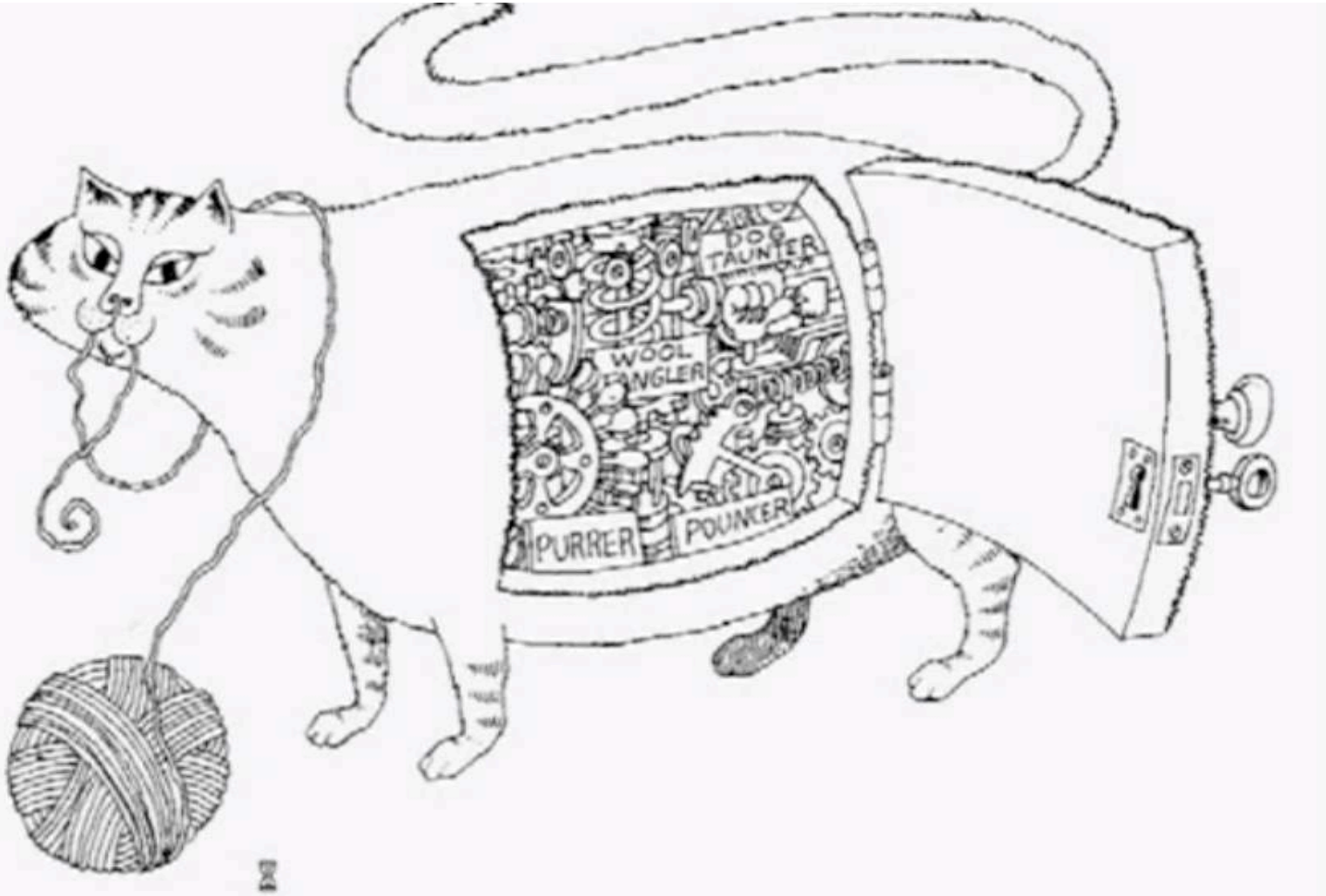




Information Hiding:
How does an object encapsulate?
What does it encapsulate?

Abstraction and encapsulation are complementary concepts: Abstraction focuses on the observable behavior of an object, whereas encapsulation focuses on the implementation that gives rise to this behavior.

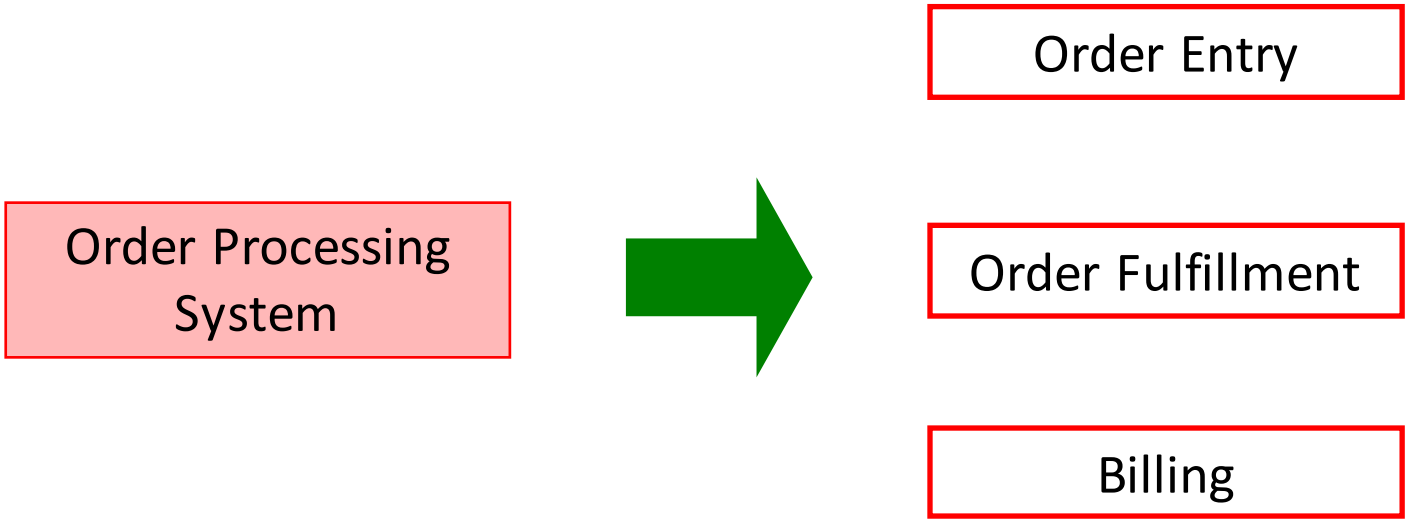
What is Encapsulation?



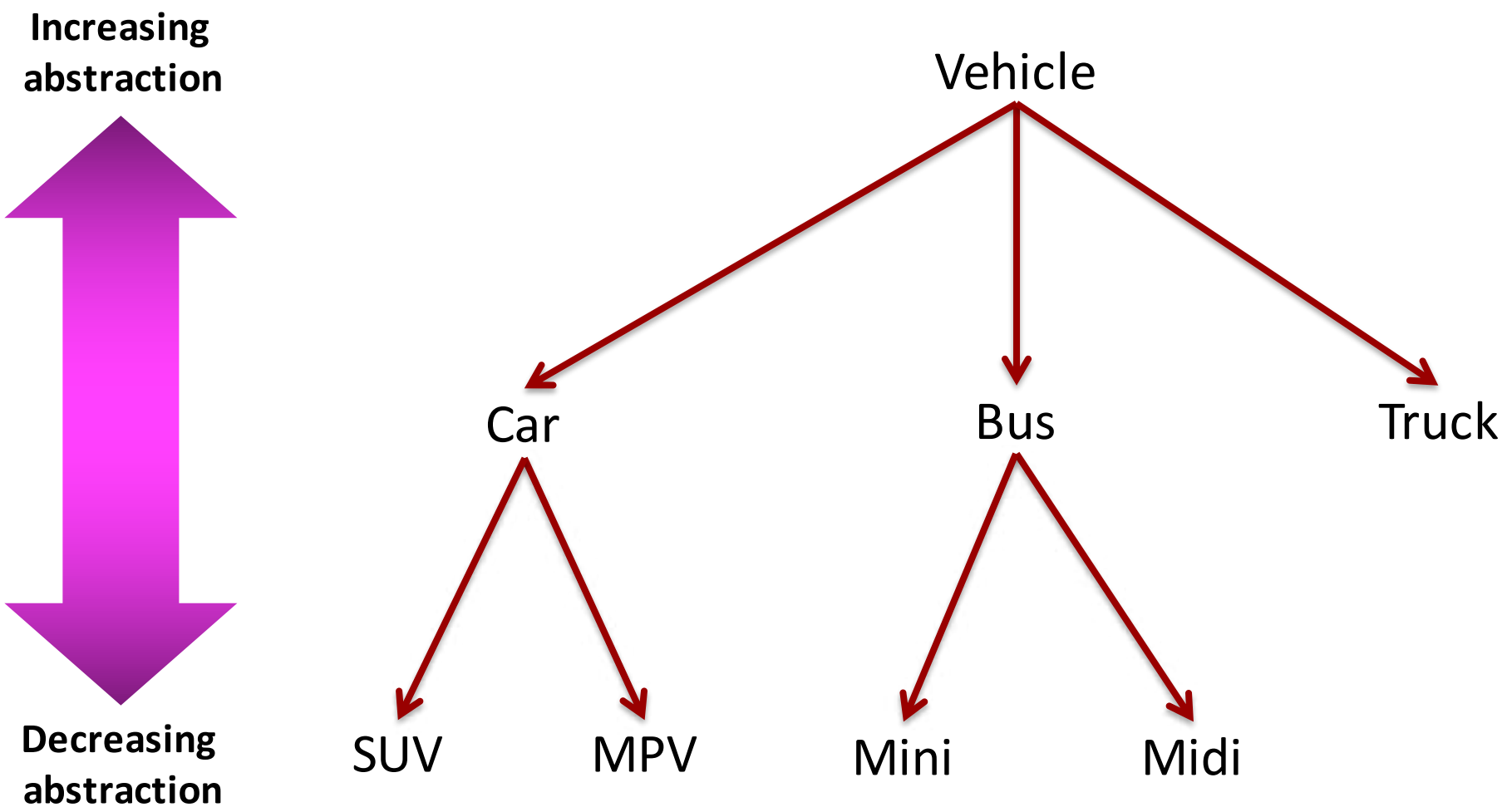
Encapsulation hides the details of the implementation of an object.

What is Modularity?

- The breaking up of something complex into manageable pieces.



What is Hierarchy?



Elements at the same level of the hierarchy should be at the same level of abstraction.

What is Really an Object?

- Formally, an object is a concept, abstraction, or thing with sharp boundaries and meaning for an application.

- An object is something that has:
  - State (property, attribute)
  - Behavior (operation, method)
  - Identity

Representing Objects

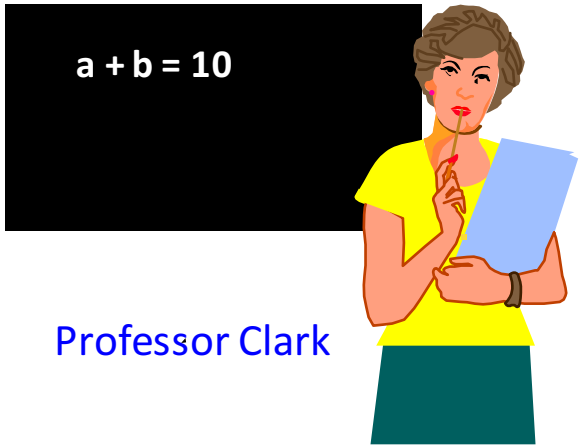
- An object is represented as rectangles with underlined names.

: Professor

Class Name Only

ProfessorClark

Object Name Only



ProfessorClark : Professor

Class and Object Name

What is a Class?

- A class is a description of a group of objects with common properties (attributes), behavior (operations), relationships, and semantics
  - An object is an instance of a class

- A class is an abstraction in that it:
  - Emphasizes relevant characteristics
  - Suppresses other characteristics

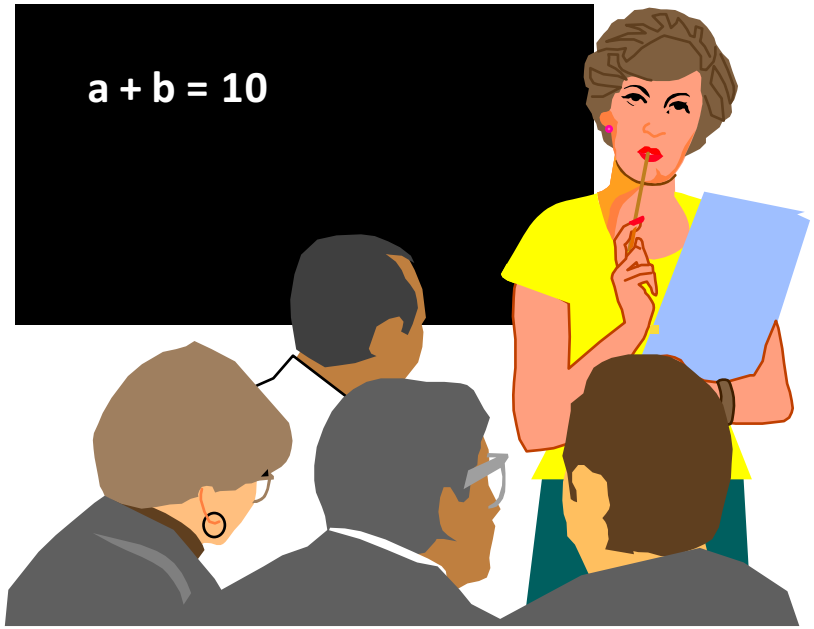
Example Class

Class

Course

Properties

Name
Location
Days offered
Credit hours
Start time
End time

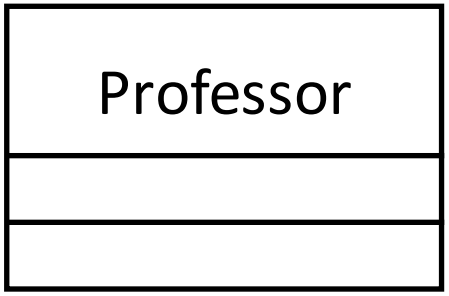


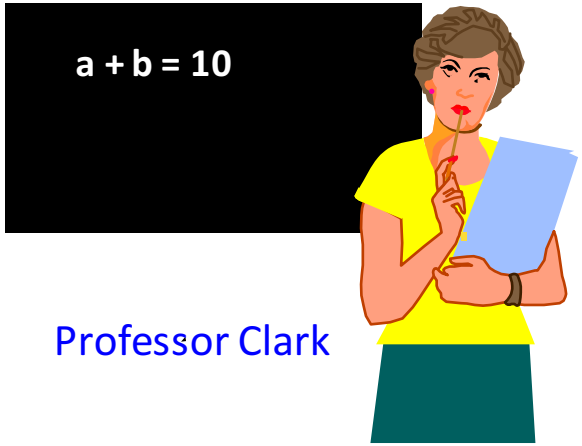
Behavior

Add a student
Delete a student
Get course roster
Determine if it is full

Representing Classes

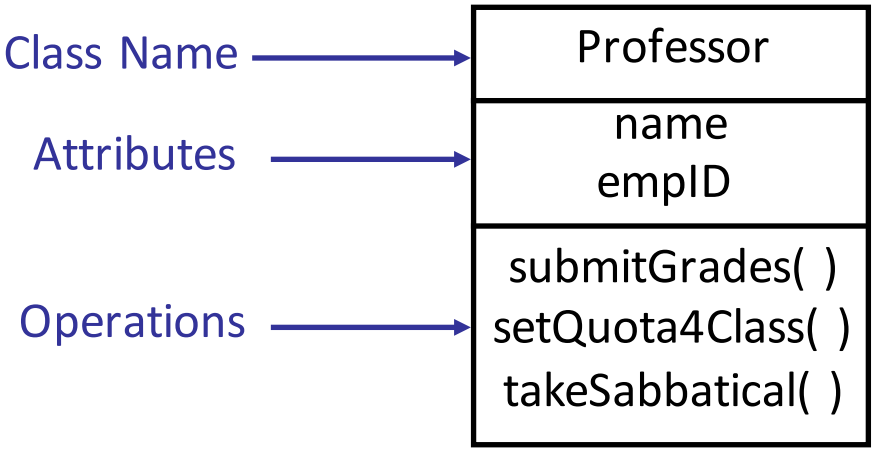
- A class is represented using a compartmented rectangle



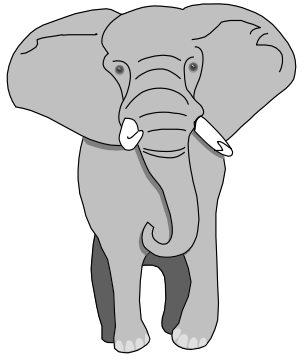


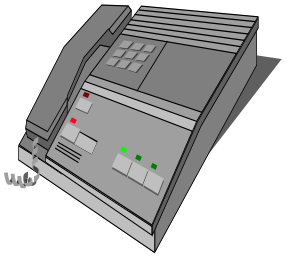
Class Compartments

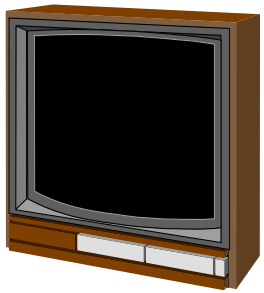
- A class is comprised of three sections
  - The first section contains the **class name**
  - The second section shows the **structure** (attributes)
  - The third section shows the **behavior** (operations)

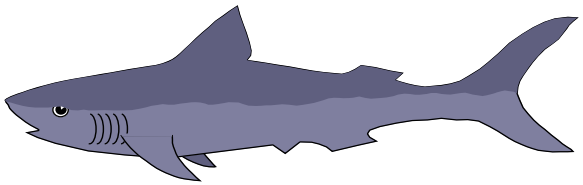


How Many Classes do you See?



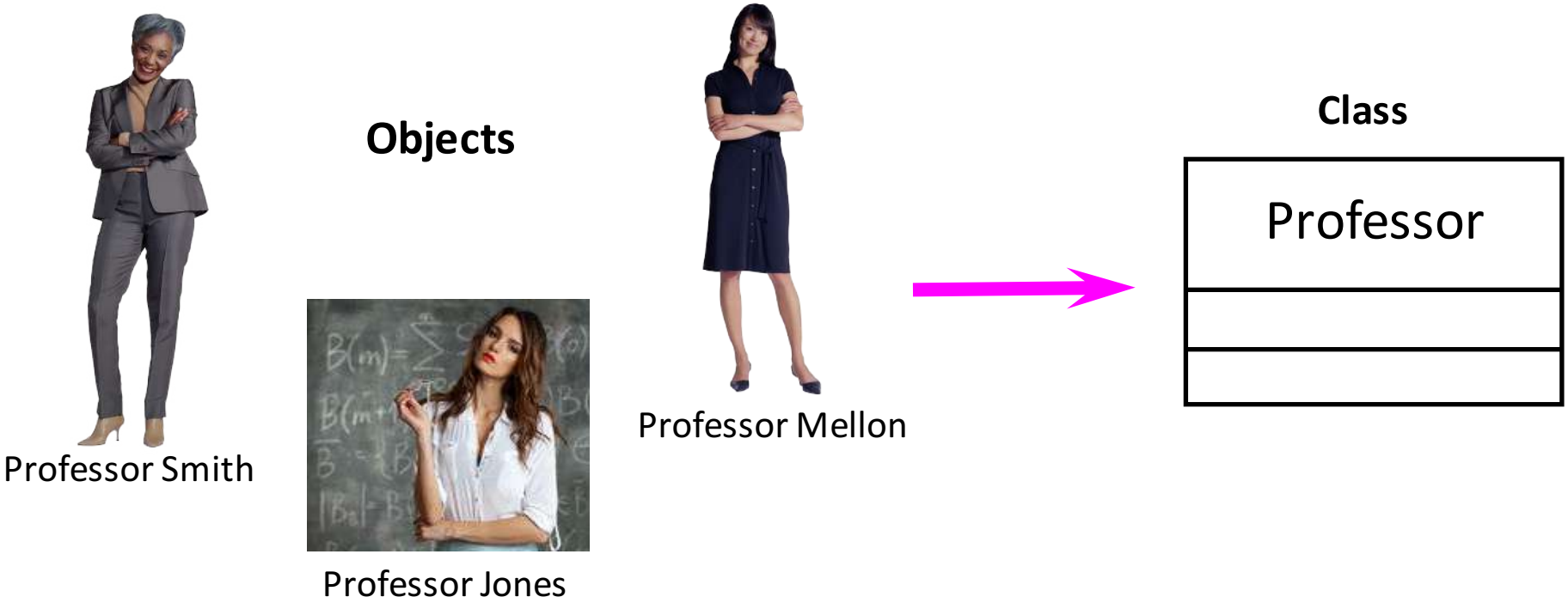






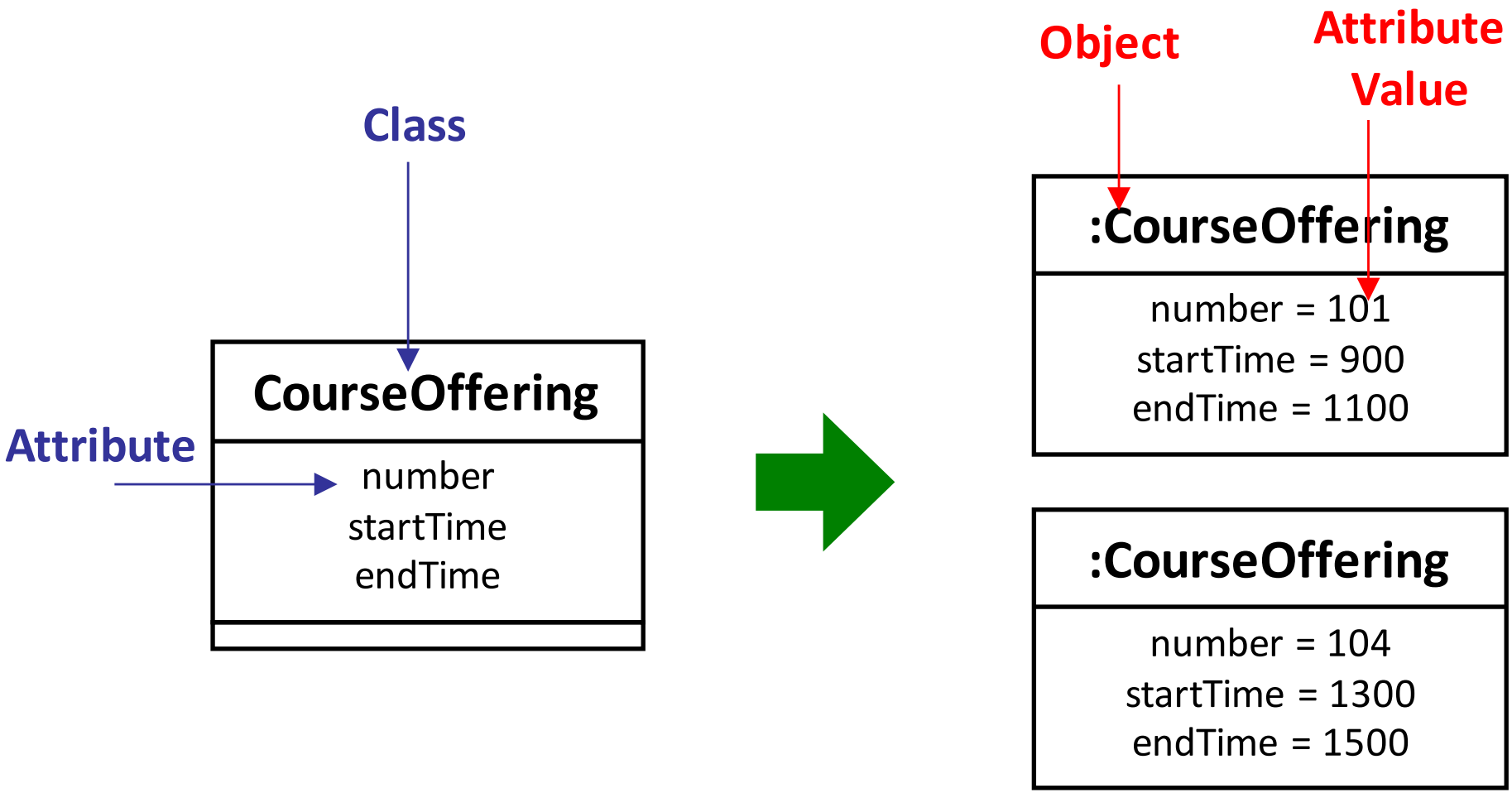
Relationship between Classes and Objects

- A class is an abstract definition of an object
  - It defines the structure and behavior of each object in the class
  - It serves as a template for creating objects
- Objects are grouped into classes



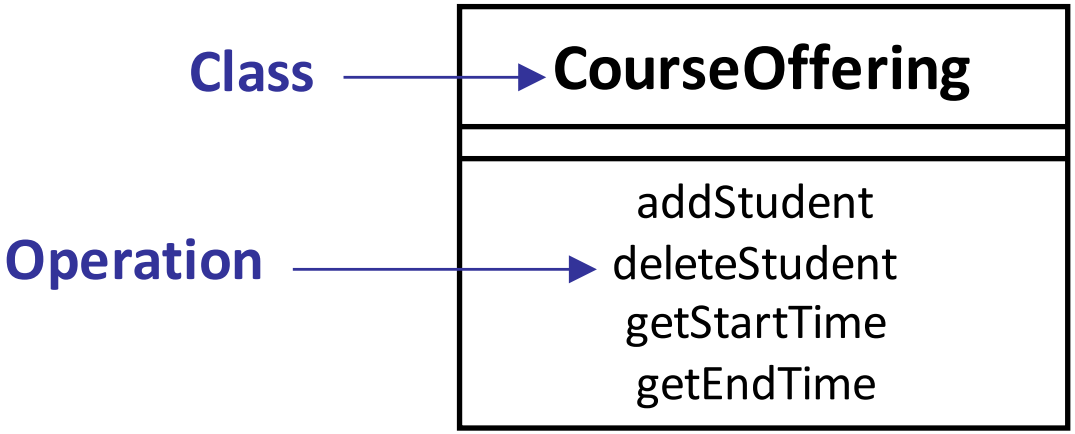
State of an Object (property or attribute)

- The state of an object encompasses all of the (usually static) properties of the object plus the current (usually dynamic) values of each of these properties.



Behavior of an Object (operation or method)

- Behavior is how an object acts and reacts, in terms of its state changes and message passing.



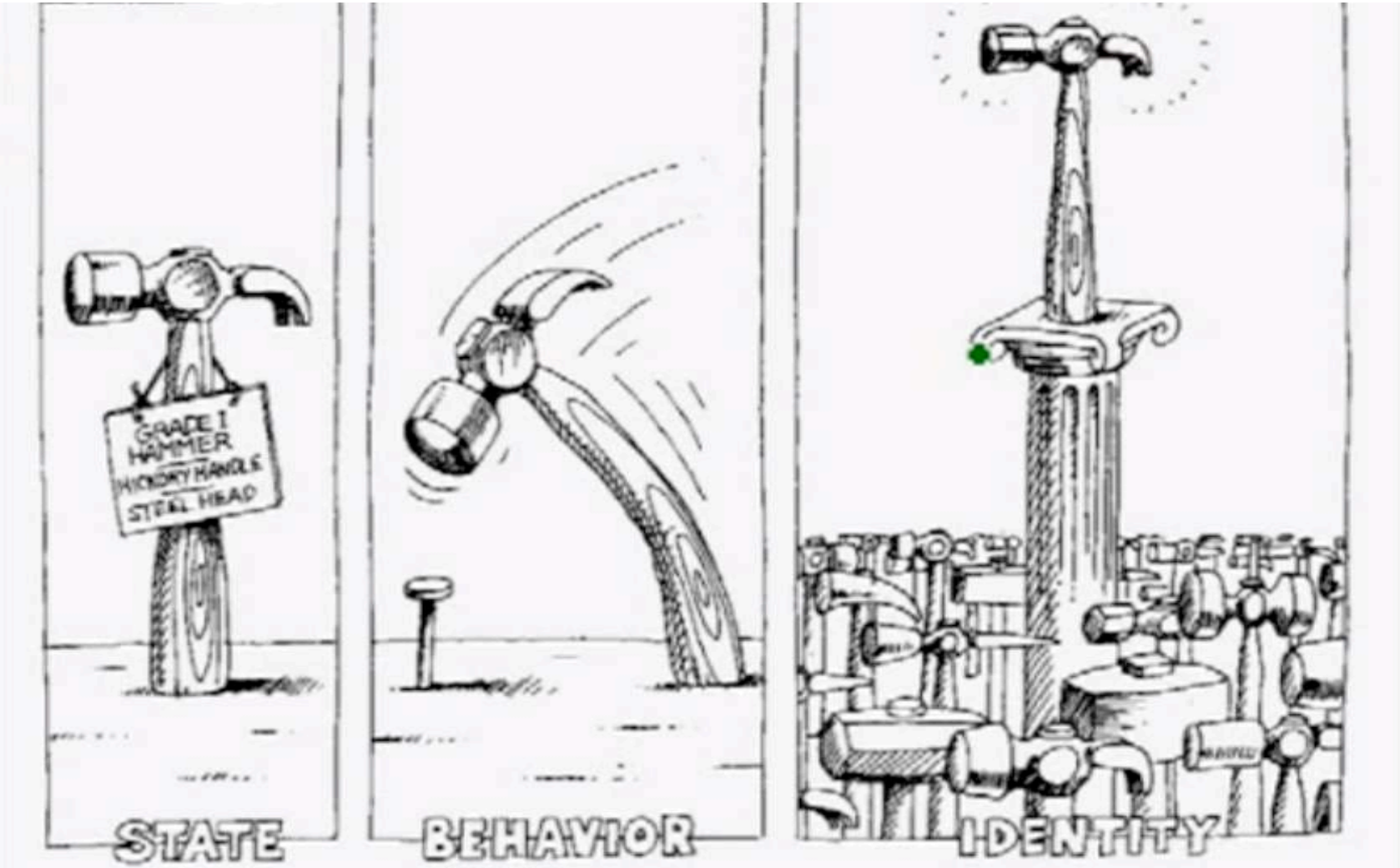
Identity of an Object

- Each object has a unique identity, even if the state is identical to that of another object.

Professor "J Clark" teaches Biology

Professor "J Clark" teaches Biology

A Class



An object has state, exhibits some well-defined behavior, and has a unique identity.

Sample Class: Automobile

- Attributes
  - manufacturer's name
  - model name
  - year made
  - color
  - number of doors
  - size of engine

- Methods
  - Define attributes (specify manufacturer's name, model, year, etc.)
  - Change a data item (color, engine, etc.)
  - Display data items
  - Calculate cost

Sample Class: Circle

- Attributes
  - Radius
  - Center Coordinates
    - X and Y values

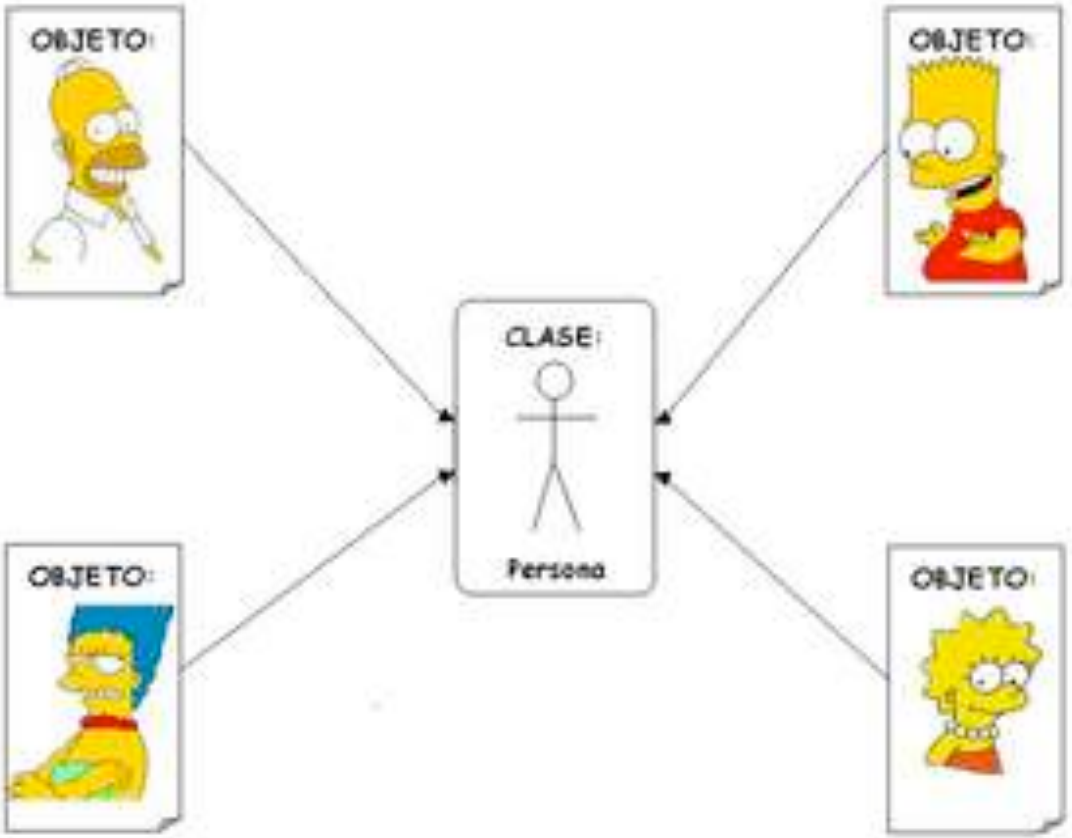
- Methods
  - Define attributes (radius and center coordinates)
  - Find area of the circle
  - Find circumference of the circle

Sample Class: Baby

- Attributes
  - Name
  - Gender
  - Weight
  - Decibel
  - # poops so far

- Methods
  - Get or Set specified attribute value
  - Poop

Sample Class: Person



Summary

- So far, we covered basics of objects and object oriented paradigm.
  - We tried to think in terms of objects.
- From now on, we should be seeing objects everywhere ☺
  - Or, we should be realizing that we were seeing objects everywhere already.
  - This is actually something you do naturally. Why not do programming that way?
- We will continue next week with actually creating objects by using Java.

Acknowledgments

- The course material used to prepare this presentation is mostly taken/adopted from the list below:
  - Software Enginering (9th ed) by I.Sommerville, Pearson, 2011.
  - Object Oriented Analysis and Design with Applications, Grady Booch, Robert A. Maksimchuk, Michael W. Engle, Bobbi J. Young, Jim Conallen and Kelli A. Houston, Addison Wesley, 2007.
  - OOAD Using the UML - Introduction to Object Orientation, v 4.2, 1998-1999 Rational Software
  - Java - An Introduction to Problem Solving and Programming, Walter Savitch, Pearson, 2012.
  - Ku-Yaw Chang, Da-Yeh University.

BBM 102 – Introduction to Programming II

Spring 2017

Classes and Objects in Java

Instructors: Ayça Tarhan, Fuat Akal, Gönenç Ercan, Vahid Garousi
TAs: Selma Dilek, Selim Yılmaz, Selman Bozkır

Today

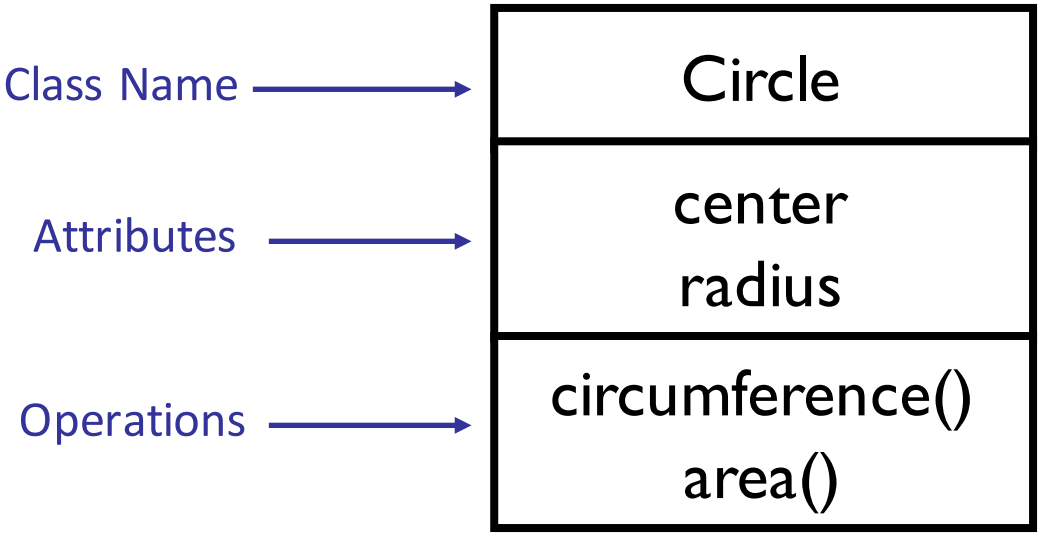
- Defining Classes, Objects and Methods
- Accessor and Mutator Methods
- Constructors
- Static Members
- Wrapper Classes
- Parameter Passing
- Delegation

Class and Method Definitions

- Java program consists of objects
  - Objects of class types
  - Objects that interact with one another
- Program objects can represent
  - Objects in real world
  - Abstractions

Java Classes

- A class is a collection of fields (data) and methods (procedure or function) that operate on that data.



Defining a Java Class

- Syntax:

```
class  ClassName{
      [fields declaration]
      [methods declaration]
}
```

- Bare bone class definition:

```
/* This is my first java class.
It is not complete yet. */
class Circle {
    // fields will come here
    // methods will come here
}
```

Adding Fields to Class Circle

- Add fields

```
class Circle {
    public double x, y; // center coordinates
    public double r;    // radius of the circle
}
```

- The fields are also called the ***instance*** variables.
  - Each object, or instance of the class has its own copy of these instance variables

- Do not worry about what *public* means at the moment.
  - Access modifiers (public, private and protected will be covered in the next weeks)

Adding Methods to a Class

- A class with only data fields has no life.
  - Objects created by such a class cannot respond to any messages.

- Methods are declared inside the body of the class.

- The general form of a method declaration is:

```
type MethodName (parameter-list)
{
      Method-body;
}
```

- methodName(parameter-list) part of the declaration is also known as the method signature.
  - Method signatures in a class should be unique!

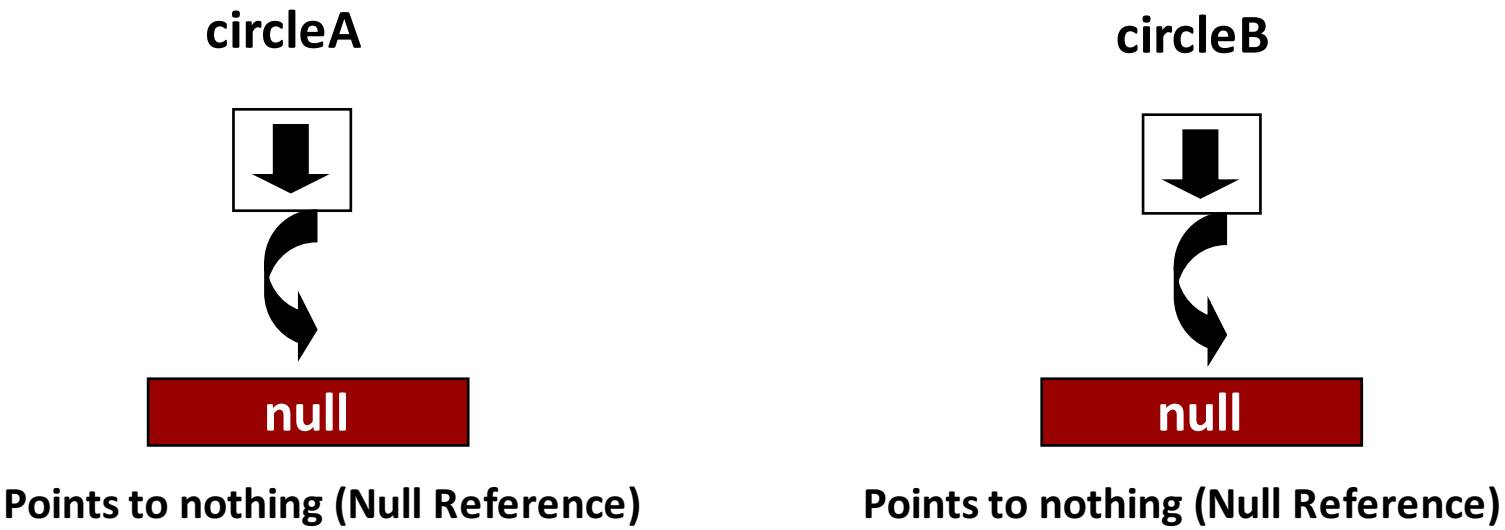
Adding Methods to Class Circle

```
public class Circle {
      public double x, y; // center of the circle
      public double r;    // radius of the circle

      // Method to return circumference
      public double circumference() {
            return 2 * 3.14 * r;
      }

      // Method to return area
      public double area() {
            return 3.14 * r * r;
      }
}
```

Defining Objects of a Class

- A class can be thought as a type
- A variable (object) can be defined as of that type (class)

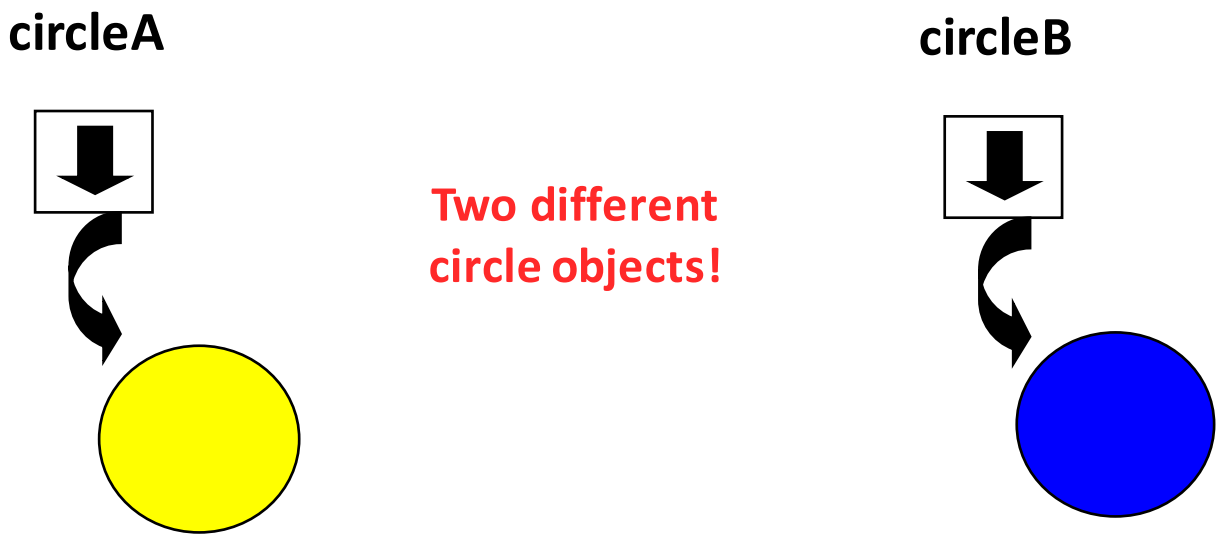
```
Circle circleA, circleB;
```



Creating Objects of a Class

- Objects are created by using the **new** keyword

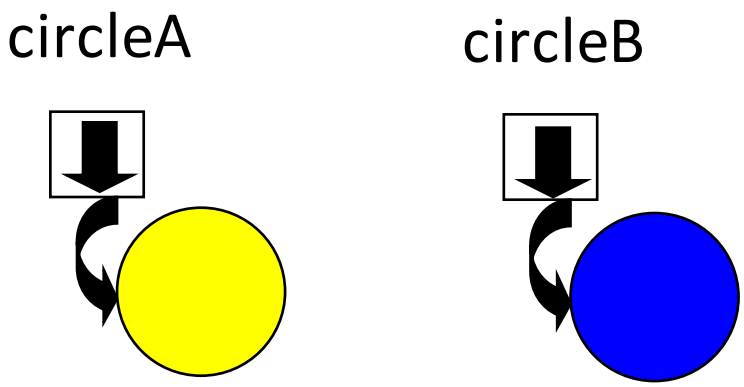
```
Circle circleA;
circleA = new Circle();

Circle circleB = new Circle();
```



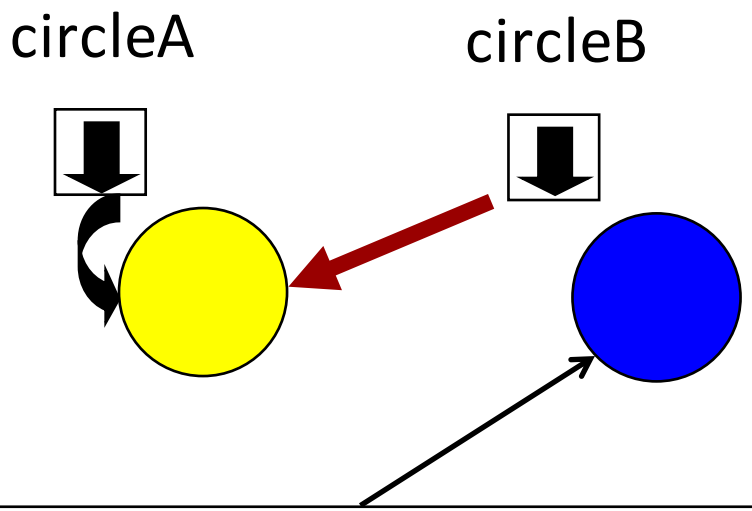
Creating Objects of a Class

```
circleA = new Circle();
circleB = new Circle() ;
circleB = circleA;
```

Before Assignment



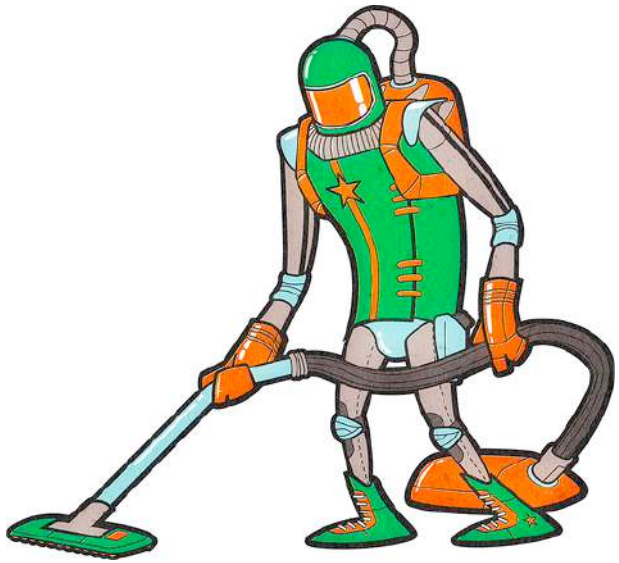
After Assignment



This object does not have a reference anymore: inaccessable!

Garbage Collection

- The object which does not have a reference cannot be used anymore.
- Such object becomes a candidate for automatic garbage collection.
- Java collects garbage periodically and releases the memory occupied by such objects to be used in the future.



Using Objects

- Object's data is accessed by using the dot notation

```
Circle circleA = new Circle();

circleA.x = 25.0;
circleA.y = 25.0;
circleA.r = 3.0;
```

- Object's methods are invoked by sending messages

```
double area = circleA.area();
```

Circle Class Alltogether

```
public class Circle {
        public double x, y; // center of the circle
        public double r;    // radius of the circle

        // Methods to return circumference and area
        public double circumference() {
                return 2 * 3.14 * r;
        }
        public double area() {
                return 3.14 * r * r;
        }
        public static void main(String[] args) {
                Circle circleA = new Circle();
                circleA.x = 25.0;
                circleA.y = 25.0;
                circleA.r = 3.0;

                double area = circleA.area();
                System.out.println("Area of the circle is " + area);
        }
}
```

Class Files and Separate Compilation

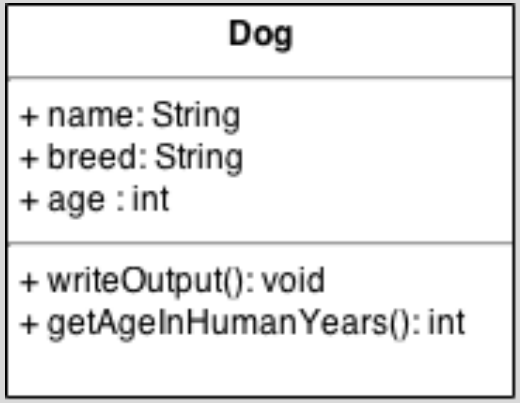
- Each Java class definition is usually written in a file by itself
  - File begins with the name of the class
  - Ends with .java
- Class can be compiled separately
- Helpful to keep all class files used by a program in the same directory

Java has Two Kinds of Methods

- Methods that <u>return a single item</u>

- Methods that <u>perform some action</u> rather than returning an item
  - void methods

```java
public class Dog {
        public String name;          // Instance variables
        public String breed;
        public int age;

        // Method that returns nothing: void method
        public void writeOutput() {
                System.out.println("Name: " + name);
                System.out.println("Breed: " + breed);
                System.out.println("Age in calendar years: " + age);
                System.out.println("Age in human years: " +
                                        getAgeInHumanYears());
        }

        // Method that returns a value
        public int getAgeInHumanYears() {
                int humanAge = 0;
                if (age <= 2) {
                        humanAge = age * 11;
                } else {
                        humanAge = 22 + ((age - 2) * 5);
                }
                return humanAge;
        }
}
```



Example Dog Class

```java
public class DogDemo {
   public static void main(String[] args) {
        Dog balto = new Dog();
        balto.name = "Balto";
        balto.age = 8;
        balto.breed = "Siberian Husky";
        balto.writeOutput();

        Dog scooby = new Dog();
        scooby.name = "Scooby";
        scooby.age = 42;
        scooby.breed = "Great Dane";
        System.out.println(scooby.name + " is a " + scooby.breed + ".");
        System.out.print("He is " + scooby.age + " years old, or ");

        int humanYears = scooby.getAgeInHumanYears();
        System.out.println(humanYears + " in human years.");
   }
}
```

DogDemo class contains only a main method.

balto:Dog
name = "Balto" breed = "Siberian Husky" age = 8

scooby:Dog
name = "Scooby" breed = "Great Dane" age = 42

Program's output

```
Name: Balto
Breed: Siberian Husky
Age in calendar years: 8
Age in human years: 52

Scooby is a Great Dane.
He is 42 years old, or 222 in human years.
```

```java
public class Dog {
        public String name;
        public String breed;
        public int age;

        public void writeOutput()  {
                // method body
        }

        public int getAgeInHumanYears()  {
                // method body
        }

        public static void main(String[] args)  {
                Dog balto = new Dog();
                balto.name = "Balto";
                balto.age = 8;
                balto.breed = "Siberian Husky";
                balto.writeOutput();
                ...
        }
}
```

Dog class could contain a main method, too.

Multiple Classes in a Single File

```
class Computer {

  void computer_method() {
    System.out.println("Power gone! Shut down your PC soon...");
  }

  public static void main(String[] args) {
    Computer my = new Computer();
    Laptop your = new Laptop();

    my.computer_method();
    your.laptop_method();
  }
}

class Laptop {
  void laptop_method() {
    System.out.println("99% Battery available.");
  }
}
```

The file *Computer.java* contains two class definitions.

```
$ javac Computer.java
// will generate Computer.class and Laptop.class files.
```

Accessor and Mutator Methods

- A public method that returns data from a private instance variable is called an accessor method, a get method, or a getter.
  - The names of accessor methods typically begin with get.

- A public method that changes the data stored in one or more private instance variables is called a mutator method, a set method, or a setter.
  - The names of mutator methods typically begin with set.

Circle Class with Getters/Setters

```
public class Circle {
      public double x, y; // center of the circle
      public double r;    // radius of the circle

      public double getX() { return x; }
      public void setX(double centerX) { x = centerX; }
      public double getY() { return y; }
      public void setY(double centerY) { y = centerY; }
      public double getR() { return r; }
      public void setR(double radius) { r = radius; }

      // Methods to return circumference and area
      …
}
```

Constructors

- Constructor is a special method that gets invoked "automatically" at the time of object creation.
- Constructor is normally used for initializing objects with default values unless different values are supplied.
- Constructor has the same name as the class name.
- Constructor cannot return values.
- A class can have more than one constructor as long as they have different signature (i.e., different input arguments syntax).

Circle Class with Constructor

```
public class Circle {
        public double x, y; // center of the circle
        public double r;    // radius of the circle

        // Constructor
        public Circle(double centerX, double centerY, double radius) {
                x = centerX;
                y = centerY;
                r = radius;
        }

        // Methods to return circumference and area
        ...
}
```

```
Circle aCircle = new Circle(10.0, 20.0, 5.0);
```

Multiple Constructors

- Sometimes we may want to initialize in a number of different ways, depending on the circumstance.

- This can be supported by having multiple constructors having different input arguments.

Circle Class with Multiple Constructors

```
public class Circle {
        public double x, y; // center of the circle
        public double r;    // radius of the circle

        // Constructor
        public Circle(double centerX, double centerY, double radius) {
                x = centerX;
                y = centerY;
                r = radius;
        }

        public Circle(double radius) {
                x = 0; y = 0; r = radius;
        }

        public Circle() {
                x = 0; y = 0; r = 1.0;
        }

        // Methods to return circumference and area
        ...
}
```

```
Circle aCircle = new Circle(10.0, 20.0, 5.0);
Circle bCircle = new Circle(5.0);
Circle cCircle = new Circle();
```

The Keyword this

- this keyword can be used to refer to the object itself.
- It is generally used for accessing class members (from its own methods) when they have the same name as those passed as arguments.

```
public class Circle {
        public double x, y; // center of the circle
        public double r;    // radius of the circle

        public double getX() { return x; }
        public void setX(double x) { this.x = x; }
        public double getY() { return y; }
        public void setY(double y) { this.y = y; }
        public double getR() { return r; }
        public void setR(double r) { this.r = r; }

        // Methods to return circumference and area
        …
}
```

Static Variables

- Java supports definition of global variables that can be accessed without creating objects of a class.
  - Such members are called Static members.

- This feature is useful when we want to create a variable common to all instances of a class.

- One of the most common example is to have a variable that could keep a count of how many objects of a class have been created.

- Java creates only one copy for a static variable which can be used even if the class is never instantiated.

Using Static Variables

- Define the variable by using the static keyword

```
public class Circle {
    // Class variable, one for the Circle class.
    // To keep number of objects created.
    public static int numCircles;

    // Instance variables, one for each instance
    // of the Circle class.
    public double x,y,r;

    // Constructor
    Circle (double x, double y, double r){
        this.x = x;
        this.y = y;
        this.r = r;
        numCircles++;
    }
}
```

```
Circle circleA = new Circle(10, 12, 20);
// numCircles = 1
Circle circleB = new Circle(5, 3, 10);
// numCircles = 2
```

Instance vs. Static Variables

- *Instance variables*: One copy per object. Every object has its own instance variables.
  - e.g. **`x,y,r`** (center and radius of the circle)

- *Static variables*: One copy per class.
  - e.g. **`numCircles`** (total number of circle objects created)

Static Methods

- A class can have methods that are defined as **static**.

- Static methods can be accessed without using objects. Also, there is **NO** need to create objects.

- Static methods are generally used to group related library functions that don't depend on data members of its class.
  - e.g., Math library functions.

Using Static Methods

```
class Comparator {
        public static int max(int a, int b) {
                if (a > b)
                        return a;
                else
                        return b;
        }

        public static String max(String a, String b) {
                if (a.compareTo(b) > 0)
                        return a;
                else
                        return b;
        }
}
```

```
// Max methods are directly accessed using ClassName.
// NO Objects created.
System.out.println(Comparator.max(5, 10));
System.out.println(Comparator.max("ANKARA", "SAMSUN"));
```

More Static Methods: The Math Class

- It is like including libraries in C language
- It contains standard mathematical methods
  - They are all static
  - Java.lang.Math

```
Math.pow(2.0, 3.0)      // 8
Math.max(5, 6)          // 6
Math.round(6.2)         // 6
Math.sqrt(4.0)          // 2.0
```

Object Cleanup (Destructor)

- Recall: Memory deallocation is automatic in Java
  - No dangling pointers and no memory leak problem.
- Java allows to define **finalize** method, which is invoked (if defined) just before the object destruction.
- This presents an opportunity to perform record maintenance operation or clean up any special allocations made by the user.
- The finalize method will be called by the Garbage Collector, but when this will happen is not deterministic. Try to avoid finalize.

```
protected void finalize() throws IOException {
   Circle.numCircles = Circle.numCircles--;
   System.out.println("Number of circles:"+ Circle.num_circles);
}
```

Wrapper Classes

- Each of Java's primitive data types has a class dedicated to it.
  - **B**oolean, **B**yte, **C**haracter, **I**nteger, **F**loat, **D**ouble, **L**ong, **S**hort
  - These are known as wrapper classes, because they "wrap" the primitive data type into an object of that class.
  - They contain useful predefined constants and methods
  - The wrapper classes are part of the java.lang package, which is imported by default into all Java programs.
  - Since Java 5.0 we have autoboxing and unboxing.

```
// Defining objects of wrapper class
Integer x = new Integer(33);
Integer y = 33; // Autoboxing
int yInt = y; // Unboxing

// Convert string to an integer
String s = "123”;
int i = Integer.parseInt(s);

//Converting from hexadecimal to decimal
Integer hex2Int = Integer.valueOf("D", 16);
```

Parameter Passing

- Method parameters which are objects are passed by reference.

- Copy of the reference to the object is passed into method, original value unchanged (e.g. circleB parameter in next slide)

```
public class ReferenceTest {

   public static void main (String[] args){
      Circle c1 = new  Circle(5, 5, 20);
      Circle c2 = new Circle(1, 1, 10);
      System.out.println ( "c1 Radius = " + c1.getRadius());
      System.out.println ( "c2 Radius = " + c2.getRadius());

      parameterTester(c1, c2);

      System.out.println ( "c1 Radius = " + c1.getRadius());
      System.out.println ( "c2 Radius = " + c2.getRadius());
   }

   public static void parameterTester(Circle circleA, Circle circleB){
      circleA.setRadius(15);
      circleB = new Circle(0, 0, 100);

      System.out.println ( "circleA Radius = " + circleA.getRadius());
      System.out.println ( "circleB Radius = " + circleB.getRadius());
   }

}
```

```
c1 Radius =  20.0
c2 Radius = 10.0
circleA Radius = 15.0
circleB Radius = 100.0
c1 Radius =  15.0
c2 Radius = 10.0
```

Delegation

- Ability for a class to delegate its responsibilities to another class.

- A way of making an object invoking services of other objects through containership.

Using Delegation

```
public class Point {
        private double xCoord;
        private double yCoord;

        public double getXCoord(){
                return xCoord;
        }
        public double getYCoord(){
                return yCoord;
        }
}
```

```
public class Circle {
        private Point center;
        public double getCenterX(){
                return center.getXCoord();              // Delegation
        }
        public double getCenterY(){
                return center.getYCoord();              // Delegation
        }
}
```

Summary

- Classes, objects, and methods are the basic components used in Java programming.
- Constructors allow seamless initialization of objects.
- Classes can have static members, which serve as global members of all objects of a class.
- Objects can be passed as parameters and they can be used for exchanging messages.
- We will continue next week with encapsulation
  - which helps in protecting data from accidental or wrong usage and also offers better security for data.

Acknowledgments

- The course material used to prepare this presentation is mostly taken/adopted from the list below:
  - Java - An Introduction to Problem Solving and Programming, Walter Savitch, Pearson, 2012.
  - Rajkumar Buyya, University of Melbourne.

BBM 102 – Introduction to Programming II

Spring 2017

Encapsulation

Instructors: Ayça Tarhan, Fuat Akal, Gönenç Ercan, Vahid Garousi
TAs: Selma Dilek, Selim Yılmaz, Selman Bozkır

Today

- Information Hiding
- Encapsulation
- Pre- and Postcondition Comments
- The public and private Modifiers
- UML Class Diagrams
- Overloading
- Packages

Information Hiding

- Programmer using a class method need <u>not</u> know details of implementation
  - Only needs to know <u>*what*</u> the method does

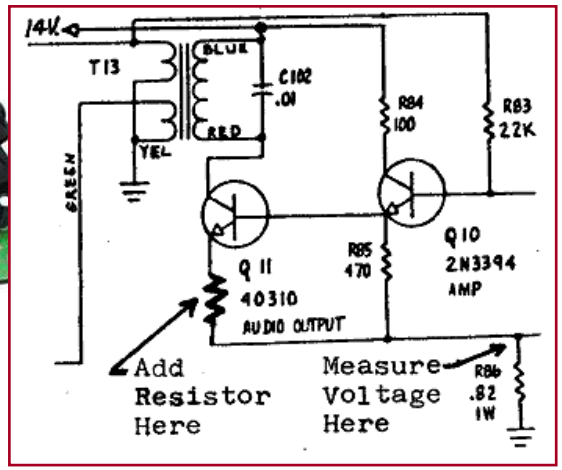
- **Information hiding:**
  - Designing a method so it can be used without knowing details

- Also referred to as ***abstraction***

- Method design should separate *what* from *how*

Encapsulation

- **Encapsulation**: Hiding implementation details of an object from its clients.
  - Encapsulation provides *abstraction*.
    - separates external view (behavior) from internal view (state)
  - Encapsulation protects the integrity of an object's data.



When Creating Classes

- When creating the public interface of a class, give careful thought and consideration to the *contract* you are creating between yourself and users (other programmers) of your class

- Use ***preconditions*** to state what you assume to be true before a method is called
  - caller of the method is responsible for making sure these are true

- Use ***postconditions*** to state what you guarantee to be true after the method is done if the preconditions are met
  - implementer of the method is responsible for making sure these are true

Pre- and Postcondition Comments

- Precondition comment
  - States conditions that must be true before method is invoked

- Example

```
/**
 Precondition: The instance variables of the calling
 object have values.
 Postcondition: The data stored in (the instance variables
 of) the receiving object have been written to the screen.
*/
public void writeOutput()
```

Pre- and Postcondition Comments

- Postcondition comment
  - Tells what will be true after method is executed

- Example

```
/**
 Precondition: years is a nonnegative number.
 Postcondition: Returns the projected population of the
 receiving object after the specified number of years.
*/
public int predictPopulation(int years)
```

Visibility Modifiers

- All parts of a *class* have visibility modifiers
  - Java keywords
  - **public**, protected, **private**
  - do not use these modifiers on local variables (syntax error)
- **public** means that constructor, method, or field <u>may be accessed</u> outside of the class.
  - part of the interface
  - constructors and methods are generally public
- **private** means that part of the class is <u>hidden and inaccessible</u> by code outside of the class
  - part of the implementation
  - data fields are generally private

The `public` and `private` Modifiers

- Type specified as `public`
  - Any other class can directly access that object by name

- Classes are generally specified as `public`

- Instance variables are usually <u>not</u> `public`
  - Instead specify as `private`

Private fields

- A field can be declared *private*.
  - No code outside the class can access or change it.

```
private type name;
```

  - Examples:

```
private int id;
private String name;
```

- Client code sees an error when accessing private fields:

```
PointMain.java:11: x has private access in Point
System.out.println("p1 is (" + p1.x + ", " + p1.y + ")");
                                   ^
```

Accessing private state

- We can provide methods to get and/or set a field's value:

```
// A "read-only" access to the x field ("accessor")
public int getX() {
    return x;
}

// Allows clients to change the x field ("mutator")
public void setX(int newX) {
    x = newX;
}
```

- Client code will look more like this:

```
System.out.println("p1: (" + p1.getX() + ", " + p1.getY() + ")");
p1.setX(14);
```

Programming Example

```
public class Rectangle
{
    private int width;
    private int height;
    private int area;

    public void setDimensions (int newWidth, int newHeight)
    {
        width = newWidth;
        height = newHeight;
        area = width * height;
    }
    public int getArea ()
    {
        return area;
    }
}
```

Note `setDimensions` method : This is the only way the `width` and `height` may be altered outside the class

- Statement such as

  `box.width = 6;`

  is illegal since width is `private`
- Keeps remaining elements of the class consistent

Point class

```java
// A Point object represents an (x, y) location.
public class Point {
    private int x;
    private int y;

    public Point(int initialX, int initialY) {
        x = initialX;
        y = initialY;
    }

    public double distanceFromOrigin() {
        return Math.sqrt(x * x + y * y);
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public void setLocation(int newX, int newY) {
        x = newX;
        y = newY;
    }

    public void translate(int dx, int dy) {
        x = x + dx;
        y = y + dy;
    }
}
```

Client code

```
public class PointMain4 {
    public static void main(String[] args) {
        // create two Point objects
        Point p1 = new Point(5, 2);
        Point p2 = new Point(4, 3);

        // print each point
        System.out.println("p1: (" + p1.getX() + ", " + p1.getY() + ")");
        System.out.println("p2: (" + p2.getX() + ", " + p2.getY() + ")");

        // move p2 and then print it again
        p2.translate(2, 4);
        System.out.println("p2: (" + p2.getX() + ", " + p2.getY() + ")");
  }
}
```

OUTPUT :

```
p1 is (5, 2)
p2 is (4, 3)
p2 is (6, 7)
```

Encapsulation

- Consider example of driving a car
  - We see and use break pedal, accelerator pedal, steering wheel – know <u>what</u> they do
  - We do <u>not</u> see mechanical details of <u>how</u> they do their jobs

- Encapsulation divides class definition into
  - *Class interface*
  - *Class implementation*

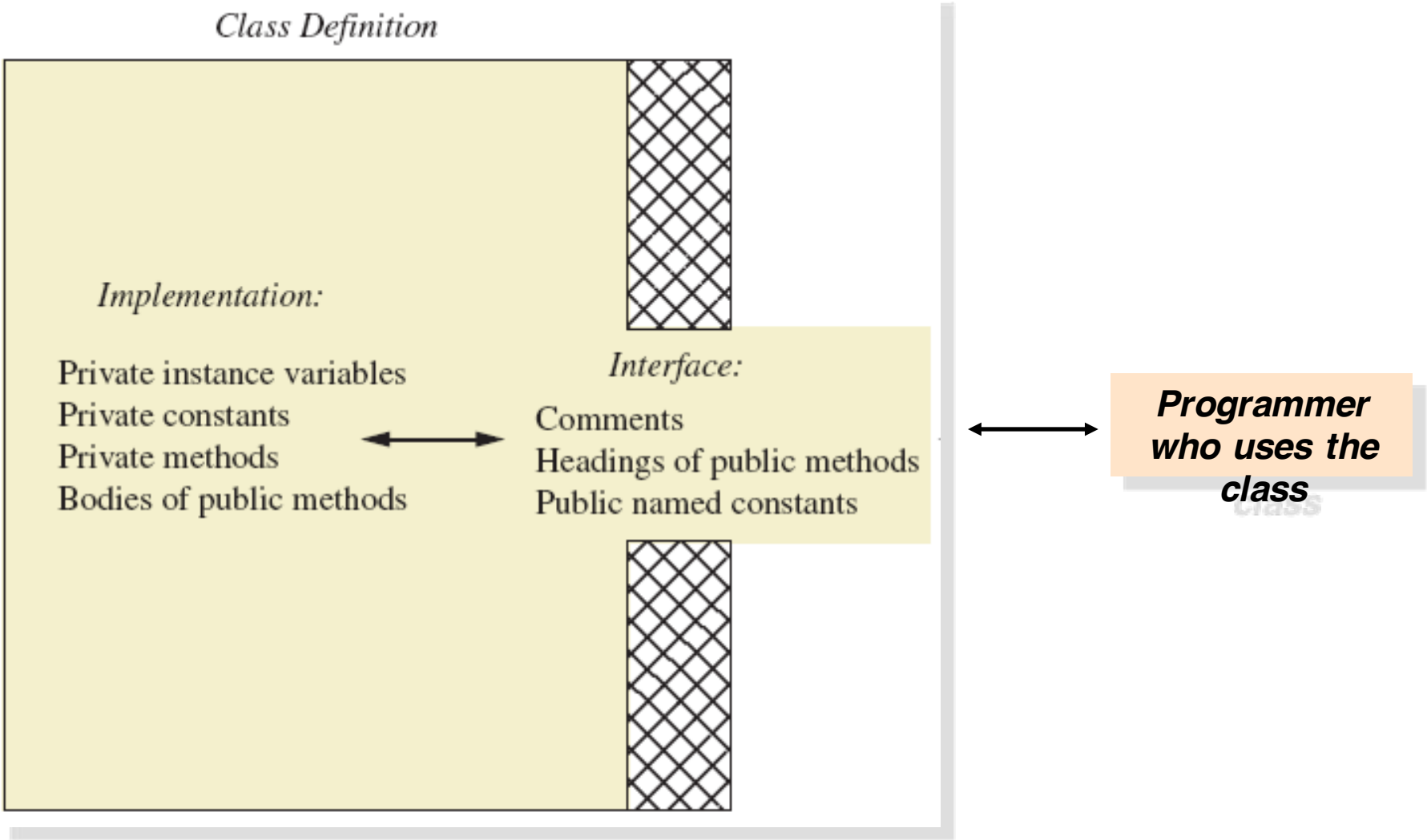
Encapsulation

- *Class interface*
  - Tells <u>what</u> the class does
  - Gives headings for public methods and comments about them

- *Class implementation*
  - Contains private variables
  - Includes definitions of public and private methods

Encapsulation

- A well encapsulated class definition



Encapsulation – Best Practices

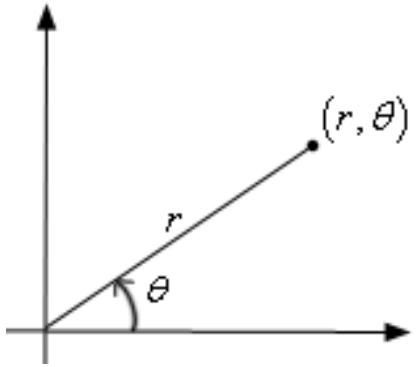
- Preface class definition with comment on how to use class
- Declare all instance variables in the class as private.
- Provide public accessor methods to retrieve data and provide public methods to manipulate data
  - Such methods could include public mutator methods.
- Place a comment before each public method heading that fully specifies how to use method.
- Make any helping methods private.
- Write comments within class definition to describe implementation details.

Benefits of encapsulation

- Provides abstraction between an object and its clients.

- Protects an object from unwanted access by clients.
  - A bank app forbids a client to change an `Account`'s balance.

- Allows you to change the class implementation.
  - `Point` could be rewritten to use polar coordinates (radius *r*, angle ϑ), but with the same methods.



- Allows you to constrain objects' state (**invariants**).
  - Example: Only allow `Points` with non-negative coordinates.

Software Development Observations

- Interfaces change less frequently than implementations.
- When an implementation changes, implementation-dependent code must change accordingly.
- Hiding the implementation reduces the possibility that other program parts will become dependent on class-implementation details.

Outline

Time1.java

(1 of 2)

```
// Fig. 8.1: Time1.java
// Time1 class declaration maintains the time in 24-hour format.

public class Time1
{
   private int hour;   // 0 - 23
   private int minute; // 0 - 59
   private int second; // 0 - 59

   // set a new time value using universal time; ensure that
   // the data remains consistent by setting invalid values to zero
   public void setTime( int h, int m, int s )

      hour = ( ( h >= 0 && h < 24 ) ? h : 0 );   // validate hour
      minute = ( ( m >= 0 && m < 60 ) ? m : 0 ); // validate minute
      second = ( ( s >= 0 && s < 60 ) ? s : 0 ); // validate second
   } // end method setTime

```

private instance variables

Declare **public** method **setTime**

Validate parameter values before setting instance variables

Outline

```
   // convert to String in universal-time format (HH:MM:SS)
   public String toUniversalString()
   {
      return String.format( "%02d:%02d:%02d", hour, minute, second );
   } // end method toUniversalString

   // convert to String in standard-time format (H:MM:SS AM or PM)
   public String toString()
   {
      return String.format( "%d:%02d:%02d %s",
         ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ),
         minute, second, ( hour < 12 ? "AM" : "PM" ) );
   } // end method toString
} // end class Time1
```

format strings

Time1.java

(2 of 2)

Outline

Time1Test.java
(1 of 2)

```java
// Fig. 8.2: Time1Test.java
// Time1 object used in an application.

public class Time1Test
{
   public static void main( String args[] )
   {
      // create and initialize a Time1 object
      Time1 time = new Time1(); // invokes Time1 constructor

      // output string representations of the time
      System.out.print( "The initial universal time is: " );
      System.out.println( time.toUniversalString() );
      System.out.print( "The initial standard time is: " );
      System.out.println( time.toString() );
      System.out.println(); // output a blank line

```

Create a **`Time1`** object

Call **`toUniversalString`** method

Call **`toString`** method

Outline

Time1Test.java

(2 of 2)

```
18        // change time and output updated time
19        time.setTime( 13, 27, 6 );
20        System.out.print( "Universal time after setTime is: " );
21        System.out.println( time.toUniversalString() );
22        System.out.print( "Standard time after setTime is: " );
23        System.out.println( time.toString() );
24        System.out.println(); // output a blank line
25
26        // set time with invalid values; output updated time
27        time.setTime( 99, 99, 99 );
28        System.out.println( "After attempting invalid settings:" );
29        System.out.print( "Universal time: " );
30        System.out.println( time.toUniversalString() );
31        System.out.print( "Standard time: " );
32        System.out.println( time.toString() );
33     } // end main
34 } // end class Time1Test
```

Call **setTime** method

Call **setTime** method with invalid values

```
The initial universal time is: 00:00:00
The initial standard time is: 12:00:00 AM

Universal time after setTime is: 13:27:06
Standard time after setTime is: 1:27:06 PM

After attempting invalid settings:
Universal time: 00:00:00
Standard time: 12:00:00 AM
```

Performance Tip

- Java conserves storage by maintaining <u>only one copy of each method per class</u>
  - this method is invoked by every object of the class.

- Each object, on the other hand, has its own copy of the class's instance variables (i.e., non-static fields).

- Each method of the class implicitly uses `this` to determine the specific object of the class to manipulate.

Software Development Observations & Tips

- When one object of a class has a reference to another object of the same class, the first object can access all the second object's data and methods (including those that are `private`).

- When implementing a method of a class, use the class's *set* and *get* methods to access the class's `private` data. This simplifies code maintenance and reduces the likelihood of errors.

- This architecture helps hide the implementation of a class from its clients, which improves program modifiability

Default and No-Argument Constructors

- Every class must have at least one constructor
  - If no constructors are declared, the compiler will create a default constructor
    - Takes no arguments and initializes instance variables to their initial values specified in their declaration or to their default values
      - Default values are `zero` for primitive numeric types, `false` for `boolean` values and `null` for references
  - If constructors are declared, the default initialization for objects of the class will be performed by a no-argument constructor (if one is declared)

Common Programming Error

- If a class has constructors, but none of the `public` constructors are no-argument constructors, and a program attempts to call a no-argument constructor to initialize an object of the class, a compilation error occurs.

- A constructor can be called with no arguments only if the class does not have any constructors (in which case the default constructor is called) or if the class has a `public` no-argument constructor.

`final` Instance Variables

- `final` instance variables
  - Keyword `final`
    - Specifies that a variable is not modifiable (is a constant)
  - `final` instance variables can be initialized at their declaration
    - If they are not initialized in their declarations, they must be initialized in all constructors

- If an instance variable should not be modified, declare it to be `final` to prevent any erroneous modification.

`static final` Instance Variables

- A `final` field should also be declared `static` if it is initialized in its declaration.
- Once a `final` field is initialized in its declaration, its value can never change.
- Therefore, it is not necessary to have a separate copy of the field for every object of the class.
- Making the field `static` enables all objects of the class to share the `final` field.
- Example: `public static final double PI = 3.141592;`

Software Reusability

- Rapid application development
  - Reusability speeds the development of powerful, high-quality software
- Java's API
  - provides an entire framework in which Java developers can work to achieve true reusability and rapid application development
  - Documentation:
    - java.sun.com/j2se/5.0/docs/api/index.html
    - Or java.sun.com/j2se/5.0/download.html to download
- **Good Programming Practice:** Avoid reinventing the wheel. Study the capabilities of the Java API. If the API contains a class that meets your program's requirements, use that class rather than create your own.

UML Class Diagrams

- An automobile class outline as a UML class diagram

```
Automobile

- fuel: double
- speed: double
- license: String

+ accelerate(double pedalPressure): void
+ decelerate(double pedalPressure): void
```

UML Class Diagrams

- Example:
  `Purchase`
  class

```
Purchase
- name: String
- groupCount: int
- groupPrice: double
- numberBought: int

+ setName(String newName): void
+ setPrice(int count, double costForCount): void
+ setNumberBought(int number): void
+ readInput( ): void
+ writeOutput( ): void
+ getName( ): String
+ getTotalCost( ): double
+ getUnitCost( ): double
+ getNumberBought( ): int
```

Minus signs imply private access

Plus signs imply public access

UML Class Diagrams

- Contains more than interface, less than full implementation
- Usually written *before* class is defined
- Used by the programmer defining the class
  - Contrast with the *interface* used by programmer who uses the class

Packages and Importing

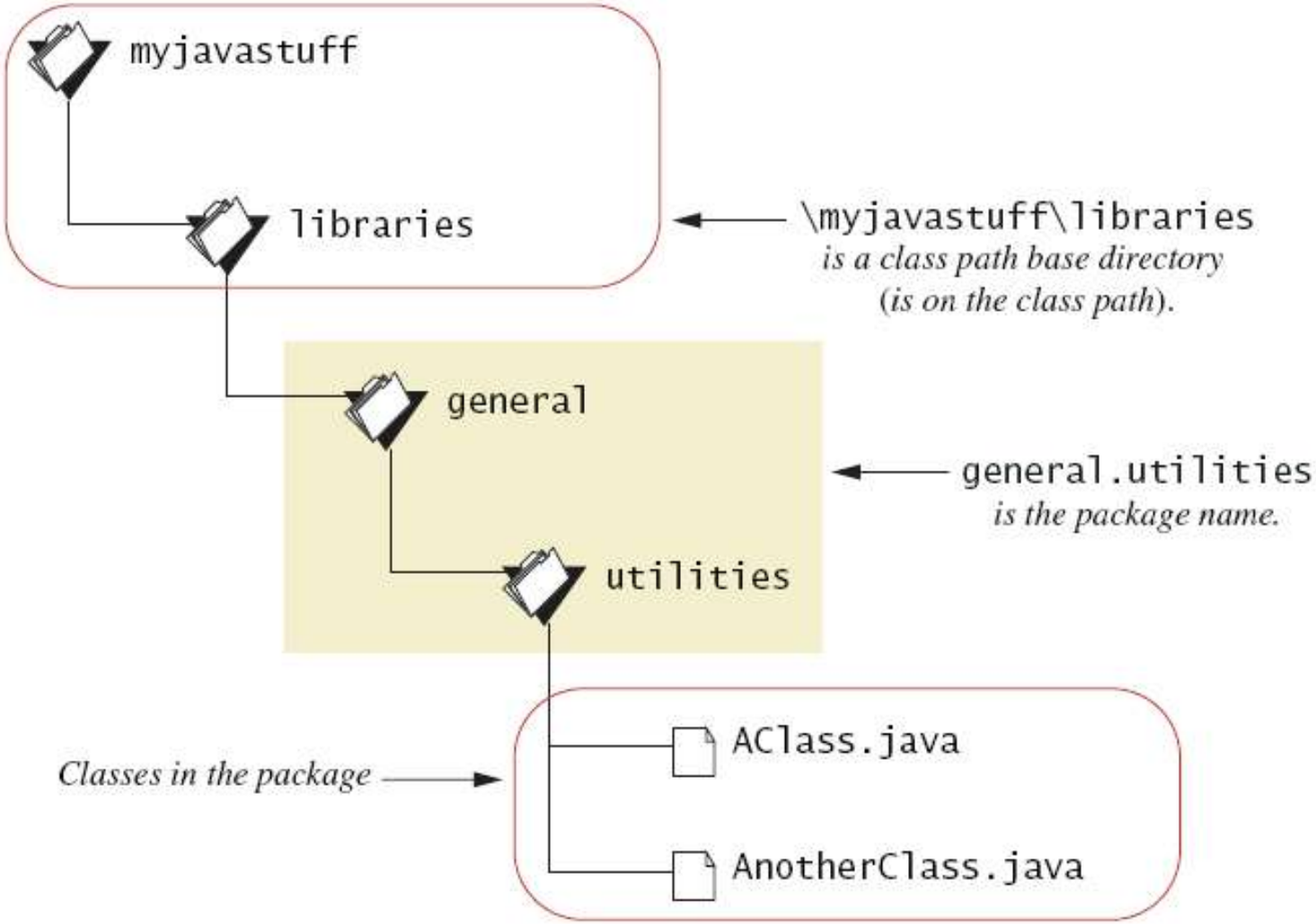
- A **package** is a collection of classes grouped together into a folder
- Name of folder is name of package
- Each class
  - Placed in a separate file
  - Has this line at the beginning of the file
    `package Package_Name;`
- Classes use packages by use of `import` statement

Package Names and Directories

- Package name tells compiler path name for directory containing classes of package
- Search for package begins in class path base directory
  - Package name uses dots in place of / or \
- Name of package uses relative path name starting from any directory in class path

Package Names and Directories

- A package name



Time Class Case Study: Creating Packages

- To declare a reusable class
  - Declare a `public` class
  - Add a `package` declaration to the source-code file
    - must be the very first executable statement in the file
      - Package name example: `com.deitel.jhtp6.ch08`
      - `package` name is part of the fully qualified class name
        - » Distinguishes between multiple classes with the same name belonging to different packages
        - » Prevents name conflict (also called name collision)

Example

- Time1.java

```java
// Fig. 8.18: Time1.java
// Time1 class declaration maintains the time in 24-hour format.
package com.deitel.jhtp6.ch08;

public class Time1
{
   private int hour;   // 0 - 23
   private int minute; // 0 - 59
   private int second; // 0 - 59

   // set a new time value using universal time; perform
   // validity checks on the data; set invalid values to zero
   public void setTime( int h, int m, int s )
   {
      hour = ( ( h >= 0 && h < 24 ) ? h : 0 );   // validate hour
      minute = ( ( m >= 0 && m < 60 ) ? m : 0 ); // validate minute
      second = ( ( s >= 0 && s < 60 ) ? s : 0 ); // validate second
   } // end method setTime
}
```

package declaration

Time1 is a **public** class so it can be used by importers of this package

Time Class Case Study: Creating Packages (Cont.)

- Compile the class so that it is placed in the appropriate package directory structure
  - Example: our package should be in the directory

```
com
  └──→ deitel
         └──→ jhtp6
               └──→ ch08
```

  - `javac` command-line option `-d`
    - `javac` creates appropriate directories based on the class's `package` declaration
    - A period (`.`) after `-d` represents the current directory

Time Class Case Study: Creating Packages (Cont.)

- Import the reusable class into a program
  - Single-type-import declaration
    - Imports a single class
    - Example: `import java.util.Random;`
  - Type-import-on-demand declaration
    - Imports all classes in a package
    - Example: `import java.util.*;`

Name Clashes

- Packages help in dealing with name clashes
  - When two classes have same name
- Different programmers may give same name to two classes
  - Ambiguity resolved by using the package name

Overloading Basics

- When two or more methods have same name within the same class
- Java distinguishes the methods by number and types of parameters
  - If it cannot match a call with a definition, it attempts to do type conversions
- A method's name and number and type of parameters is called the *signature*

Programming Example

```
/** This class illustrates overloading. */
public class Overload {

public static void main (String [] args) {
   double average1 = Overload.getAverage (40.0, 50.0);
   double average2 = Overload.getAverage (1.0, 2.0, 3.0);
   char average3 = Overload.getAverage ('a', 'c');
       System.out.println ("average1 = " + average1);
       System.out.println ("average2 = " + average2);
       System.out.println ("average3 = " + average3); }

public static double getAverage (double first, double second) {
       return (first + second) / 2.0; }

public static double getAverage (double first, double second,
double third) { return (first + second + third) / 3.0; }

public static char getAverage (char first, char second) {
       return (char) (((int) first + (int) second) / 2); }
}
```

```
average1= 45.0
average2= 2.0
average3 = b
```

Overloading and Type Conversion

- Overloading and automatic type conversion can conflict
- Remember the compiler attempts to overload before it does type conversion
- Use descriptive method names, avoid overloading when possible

Overloading and Return Type

- You must not overload a method where the only difference is the type of value returned

```
/**
 Returns the weight of the pet.
*/
public double getWeight()

/**
 Returns '+' if overweight, '-' if
 underweight, and '*' if weight is OK.
*/
public char getWeight()
```

Summary

- Precondition comment states conditions that must be true before method invoked
- Postcondition comment describes resulting effects of method execution
- Usage of visibility modifiers for encapsulation
- Separation of interface and implementation is important
- Class designers use UML notation to describe classes
- Use packages for software reusability
- Overloading must be done with care

Acknowledgments

- The course material used to prepare this presentation is mostly taken/adopted from the list below:
  - Java - An Introduction to Problem Solving and Programming, Walter Savitch, Pearson, 2012
  - Java - How to Program, Paul Deitel and Harvey Deitel, Prentice Hall, 2012
  - Mike Scott, CS314 Course notes, University of Texas Austin

BBM 102 – Introduction to Programming II

Spring 2017

Inheritance

Instructors: Ayça Tarhan, Fuat Akal, Gönenç Ercan, Vahid Garousi
TAs: Selma Dilek, Selim Yılmaz, Selman Bozkır

Today

- Inheritance
- Notion of subclasses and superclasses
- protected members
- UML Class Diagrams for inheritance

Inheritance

- A form of *software reuse* in which a new class is created by absorbing an existing class's members and embellishing them with new or modified capabilities.

- Can save time during program development by basing new classes on existing proven and debugged high-quality software.

- Increases the likelihood that a system will be implemented and maintained effectively.

Inheritance

- When creating a class, rather than declaring completely new members, you can designate that the new class should inherit the members of an existing class.
  - Existing class is the *superclass*
  - New class is the *subclass*

- The subclass exhibits the behaviors of its superclass and can add behaviors that are specific to the subclass.
  - This is why inheritance is sometimes referred to as *specialization*.

- A subclass is more specific than its superclass and represents a more specialized group of objects.

Inheritance

- The *direct superclass* is the superclass from which the subclass explicitly inherits.
- An *indirect superclass* is any class above the direct superclass in the *class hierarchy*.
- The Java class hierarchy begins with class Object (in package java.lang)
  - *Every* class in Java directly or indirectly extends (or "inherits from") Object.
- Java supports only *single inheritance*, in which each class is derived from exactly one direct superclass.

© Copyright 1992-2012 by Pearson Education, Inc. All Rights Reserved.

Advantages of inheritance

- When a class inherits from another class, there are **three** benefits:

  (1) You can *reuse* the methods and data of the existing class

  (2) You can *extend* the existing class by adding new data and new methods

  (3) You can *modify* the existing class by overloading its methods with your own implementations

Relationships between classes

- We distinguish between the *is-a relationship* and the *has-a relationship*

- *Is-a* represents inheritance
  - In an *is-a* relationship, an object of a subclass can also be treated as an object of its superclass

- *Has-a* represents composition
  - In a *has-a* relationship, an object contains as members references to other objects

© Copyright 1992-2012 by Pearson Education, Inc. All Rights Reserved.

Superclasses and Subclasses

Superclass	Subclasses
Student	GraduateStudent, UndergraduateStudent
Shape	Circle, Triangle, Rectangle, Sphere, Cube
Loan	CarLoan, HomeImprovementLoan, MortgageLoan
Employee	Faculty, Staff
BankAccount	CheckingAccount, SavingsAccount

Fig. 9.1 | Inheritance examples.

- Superclasses tend to be “more general” and subclasses “more specific.”

© Copyright 1992-2012 by Pearson Education, Inc. All Rights Reserved.

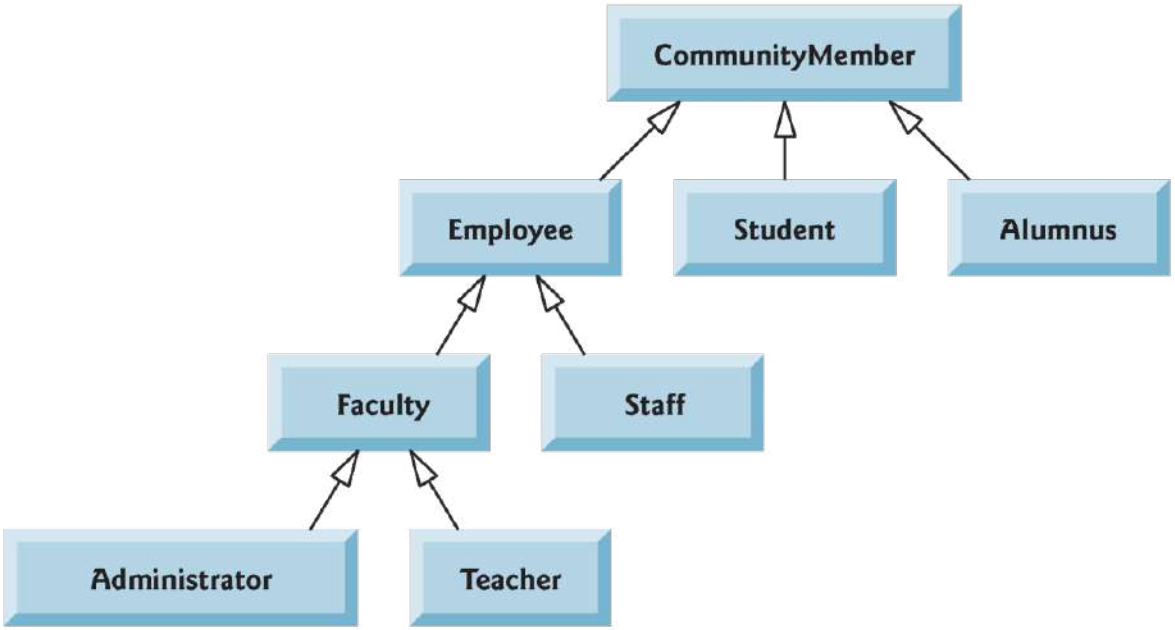


Fig. 9.2 | Inheritance hierarchy for university `CommunityMembers`.

- A sample university community class hierarchy
  - Also called an *inheritance hierarchy*.
- Each arrow in the hierarchy represents an *is-a relationship.*
- Follow the arrows upward in the class hierarchy
  - "an Employee *is a* CommunityMember"
  - "a Teacher *is a* Faculty member."

© Copyright 1992-2012 by Pearson Education, Inc. All Rights Reserved.

Superclasses and Subclasses (Cont.)

- Below is Shape inheritance hierarchy.
- Follow the arrows from the bottom of the diagram to the topmost superclass to identify several *is-a* relationships.
  - A Triangle *is a* TwoDimensionalShape and *is a* Shape
  - A Sphere *is a* ThreeDimensionalShape and *is a* Shape.

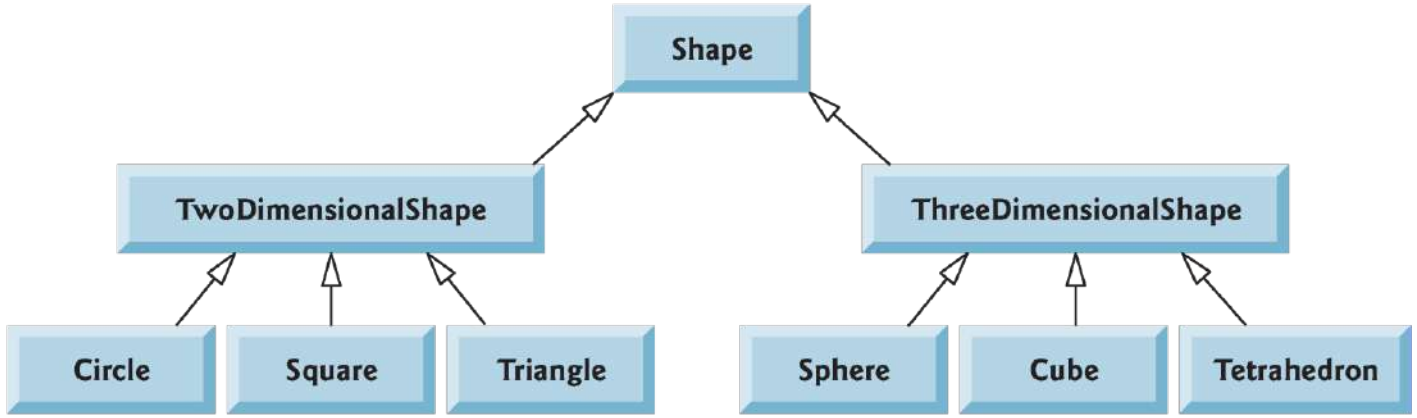


Fig. 9.3 | Inheritance hierarchy for Shapes.

© Copyright 1992-2012 by Pearson Education, Inc. All Rights Reserved.

Superclasses and Subclasses (Cont.)

- Not every class relationship is an inheritance relationship.

- *Has-a* relationship
  - Create classes by composition of existing classes.
  - Example: Given the classes `Employee`, `BirthDate` and `TelephoneNumber`, it's improper to say that an `Employee` *is a* `BirthDate` or that an `Employee` *is a* `TelephoneNumber`.
  - However, an `Employee` *has a* `BirthDate`, and an `Employee` *has a* `TelephoneNumber`.

© Copyright 1992-2012 by Pearson Education, Inc. All Rights Reserved.

`protected` Members

- A class's **`public`** members are accessible wherever the program has a reference to an object of that class or one of its subclasses.
- A class's **`private`** members are accessible only within the class itself.
- **`protected`** access is an intermediate level of access between public and private.
  - A superclass's protected members can be accessed by members of that superclass, by members of its subclasses and by members of other classes in the same package
  - protected members also have package access.

© Copyright 1992-2012 by Pearson Education, Inc. All Rights Reserved.

`protected` Members (Cont.)

- A superclass's private members are hidden in its subclasses
  - They can be accessed only through the public or protected methods inherited from the superclass
- Subclass methods can refer to public and protected members inherited from the superclass simply by using the member names.
- When a subclass method overrides an inherited superclass method, the superclass method can be accessed from the subclass by preceding the superclass method name with keyword *super* and a dot (.) separator.

© Copyright 1992-2012 by Pearson Education, Inc. All Rights Reserved.

Case Study: Commission Employees

- Inheritance hierarchy containing types of employees in a company's payroll application
- Commission employees are paid a percentage of their sales
- Base-salaried commission employees receive a base salary plus a percentage of their sales.

© Copyright 1992-2012 by Pearson Education, Inc. All Rights Reserved.

Creating and Using a `CommissionEmployee` Class

```java
// Fig. 9.4: CommissionEmployee.java
// CommissionEmployee class represents an employee paid a
// percentage of gross sales.
public class CommissionEmployee extends Object
{
   private String firstName;
   private String lastName;
   private String socialSecurityNumber;
   private double grossSales; // gross weekly sales
   private double commissionRate; // commission percentage

   // five-argument constructor
   public CommissionEmployee( String first, String last, String ssn,
      double sales, double rate )
   {
      // implicit call to Object constructor occurs here
      firstName = first;
      lastName = last;
      socialSecurityNumber = ssn;
      setGrossSales( sales ); // validate and store gross sales
      setCommissionRate( rate ); // validate and store commission rate
   } // end five-argument CommissionEmployee constructor
```

Class CommissionEmployee extends class Object (from package java.lang).

Fig. 9.4 | `CommissionEmployee` class represents an employee paid a percentage of gross sales. (Part 1 of 6.)

- `CommissionEmployee` inherits `Object`'s methods.
- If you don't explicitly specify which class a new class extends, the class extends `Object` implicitly.

```java

    // set first name
    public void setFirstName( String first )
    {
       firstName = first; // should validate
    } // end method setFirstName

    // return first name
    public String getFirstName()
    {
       return firstName;
    } // end method getFirstName

    // set last name
    public void setLastName( String last )
    {
       lastName = last; // should validate
    } // end method setLastName

```

Fig. 9.4 | `CommissionEmployee` class represents an employee paid a percentage of gross sales. (Part 2 of 6.)

```
    // return last name
    public String getLastName()
    {
       return lastName;
    } // end method getLastName

    // set social security number
    public void setSocialSecurityNumber( String ssn )
    {
       socialSecurityNumber = ssn; // should validate
    } // end method setSocialSecurityNumber

    // return social security number
    public String getSocialSecurityNumber()
    {
       return socialSecurityNumber;
    } // end method getSocialSecurityNumber

```

Fig. 9.4 | `CommissionEmployee` class represents an employee paid a percentage of gross sales. (Part 3 of 6.)

```
   // set gross sales amount
   public void setGrossSales( double sales )
   {
      if ( sales >= 0.0 )
         grossSales = sales;
      else
         throw new IllegalArgumentException(
            "Gross sales must be >= 0.0" );
   } // end method setGrossSales

   // return gross sales amount
   public double getGrossSales()
   {
      return grossSales;
   } // end method getGrossSales

```

Fig. 9.4 | `CommissionEmployee` class represents an employee paid a percentage of gross sales. (Part 4 of 6.)

```java
    // set commission rate
    public void setCommissionRate( double rate )
    {
       if ( rate > 0.0 && rate < 1.0 )
          commissionRate = rate;
       else
          throw new IllegalArgumentException(
             "Commission rate must be > 0.0 and < 1.0" );
    } // end method setCommissionRate

    // return commission rate
    public double getCommissionRate()
    {
       return commissionRate;
    } // end method getCommissionRate

    // calculate earnings
    public double earnings()
    {
       return commissionRate * grossSales;
    } // end method earnings

```

Fig. 9.4 | `CommissionEmployee` class represents an employee paid a percentage of gross sales. (Part 5 of 6.)

```java
    // return String representation of CommissionEmployee object
    @Override // indicates that this method overrides a superclass method
    public String toString()
    {
       return String.format( "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",
          "commission employee", firstName, lastName,
          "social security number", socialSecurityNumber,
          "gross sales", grossSales,
          "commission rate", commissionRate );
    } // end method toString
 } // end class CommissionEmployee
```

Fig. 9.4 | `CommissionEmployee` class represents an employee paid a percentage of gross sales. (Part 6 of 6.)

Creating and Using a CommissionEmployee Class (Cont.)

- Constructors are not inherited.
- The first task of a subclass constructor is to call its direct superclass's constructor explicitly or implicitly
  - Ensures that the instance variables inherited from the superclass are initialized properly.
- If the code does not include an explicit call to the superclass constructor, Java implicitly calls the superclass's default or no-argument constructor.
- A class's default constructor calls the superclass's default or no-argument constructor.

Creating and Using a CommissionEmployee Class (Cont.)

- **toString** is one of the methods that every class inherits directly or indirectly from class **Object**.
  - Returns a String representing an object.
  - Called implicitly whenever an object must be converted to a String representation.

- Class Object's **toString** method returns a String that includes the name of the object's class.
  - This is primarily a placeholder that can be overridden by a subclass to specify an appropriate String representation.

Creating and Using a CommissionEmployee Class (Cont.)

- To override a superclass method, a subclass must declare a method with the same signature as the superclass method

- `@Override` annotation
  - Indicates that a method should override a superclass method with the same signature.
  - If it does not, a compilation error occurs.

Common Programming Error 9.1

Using an incorrect method signature when attempting to override a superclass method causes an unintentional method overload that can lead to subtle logic errors.

Error-Prevention Tip 9.1

Declare overridden methods with the `@Override` *annotation to ensure at compilation time that you defined their signatures correctly. It's always better to find errors at compile time rather than at runtime.*

```java
// Fig. 9.5: CommissionEmployeeTest.java
// CommissionEmployee class test program.

public class CommissionEmployeeTest
{
   public static void main( String[] args )
   {
      // instantiate CommissionEmployee object
      CommissionEmployee employee = new CommissionEmployee(
         "Sue", "Jones", "222-22-2222", 10000, .06 );

      // get commission employee data
      System.out.println(
         "Employee information obtained by get methods: \n" );
      System.out.printf( "%s %s\n", "First name is",
         employee.getFirstName() );
      System.out.printf( "%s %s\n", "Last name is",
         employee.getLastName() );
      System.out.printf( "%s %s\n", "Social security number is",
         employee.getSocialSecurityNumber() );
      System.out.printf( "%s %.2f\n", "Gross sales is",
         employee.getGrossSales() );
      System.out.printf( "%s %.2f\n", "Commission rate is",
         employee.getCommissionRate() );
```

Fig. 9.5 | CommissionEmployee class test program. (Part 1 of 2.)

```java

        employee.setGrossSales( 500 ); // set gross sales
        employee.setCommissionRate( .1 ); // set commission rate

        System.out.printf( "\n%s:\n\n%s\n",
           "Updated employee information obtained by toString", employee );
    } // end main
} // end class CommissionEmployeeTest
```

Implicit `toString` call occurs here

```
Employee information obtained by get methods:

First name is Sue
Last name is Jones
Social security number is 222-22-2222
Gross sales is 10000.00
Commission rate is 0.06

Updated employee information obtained by toString:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 500.00
commission rate: 0.10
```

Fig. 9.5 | `CommissionEmployee` class test program. (Part 2 of 2.)

Case Study Part 2: Creating and Using a BasePlus-CommissionEmployee Class

- Class `BasePlusCommissionEmployee` contains a first name, last name, social security number, gross sales amount, commission rate and base salary.
  - All but the base salary are in common with class `CommissionEmployee`.

- Class `BasePlusCommissionEmployee`'s `public` services include a constructor, and methods `earnings`, `toString` and *get* and *set* for each instance variable
  - Most of these are in common with class `CommissionEmployee`.

```java
// Fig. 9.6: BasePlusCommissionEmployee.java
// BasePlusCommissionEmployee class represents an employee who receives
// a base salary in addition to commission.

public class BasePlusCommissionEmployee
{
   private String firstName;
   private String lastName;
   private String socialSecurityNumber;
   private double grossSales; // gross weekly sales
   private double commissionRate; // commission percentage
   private double baseSalary; // base salary per week

   // six-argument constructor
   public BasePlusCommissionEmployee( String first, String last,
      String ssn, double sales, double rate, double salary )
   {
      // implicit call to Object constructor occurs here
      firstName = first;
      lastName = last;
      socialSecurityNumber = ssn;
      setGrossSales( sales ); // validate and store gross sales
```

Class BasePlusCommissionEmployee does not specify "extends Object", Implicitly extends Object.

BasePlusCommissionEmployee's constructor invokes class Object's default constructor implicitly.

Fig. 9.6 | BasePlusCommissionEmployee class represents an employee who receives a base salary in addition to a commission. (Part 1 of 7.)

```
        setCommissionRate( rate ); // validate and store commission rate
        setBaseSalary( salary ); // validate and store base salary
    } // end six-argument BasePlusCommissionEmployee constructor

    // set first name
    public void setFirstName( String first )
    {
        firstName = first; // should validate
    } // end method setFirstName

    // return first name
    public String getFirstName()
    {
        return firstName;
    } // end method getFirstName

    // set last name
    public void setLastName( String last )
    {
        lastName = last; // should validate
    } // end method setLastName

```

Fig. 9.6 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 2 of 7.)

```
    // return last name
    public String getLastName()
    {
       return lastName;
    } // end method getLastName

    // set social security number
    public void setSocialSecurityNumber( String ssn )
    {
       socialSecurityNumber = ssn; // should validate
    } // end method setSocialSecurityNumber

    // return social security number
    public String getSocialSecurityNumber()
    {
       return socialSecurityNumber;
    } // end method getSocialSecurityNumber

```

Fig. 9.6 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 3 of 7.)

```
   // set gross sales amount
   public void setGrossSales( double sales )
   {
      if ( sales >= 0.0 )
         grossSales = sales;
      else
         throw new IllegalArgumentException(
            "Gross sales must be >= 0.0" );
   } // end method setGrossSales

   // return gross sales amount
   public double getGrossSales()
   {
      return grossSales;
   } // end method getGrossSales

```

Fig. 9.6 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 4 of 7.)

```
   // set commission rate
   public void setCommissionRate( double rate )
   {
      if ( rate > 0.0 && rate < 1.0 )
         commissionRate = rate;
      else
         throw new IllegalArgumentException(
            "Commission rate must be > 0.0 and < 1.0" );
   } // end method setCommissionRate

   // return commission rate
   public double getCommissionRate()
   {
      return commissionRate;
   } // end method getCommissionRate

```

Fig. 9.6 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 5 of 7.)

```java
   // set base salary
   public void setBaseSalary( double salary )
   {
      if ( salary >= 0.0 )
         baseSalary = salary;
      else
         throw new IllegalArgumentException(
            "Base salary must be >= 0.0" );
   } // end method setBaseSalary

   // return base salary
   public double getBaseSalary()
   {
      return baseSalary;
   } // end method getBaseSalary

   // calculate earnings
   public double earnings()
   {
      return baseSalary + ( commissionRate * grossSales );
   } // end method earnings

```

Fig. 9.6 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 6 of 7.)

```java
    // return String representation of BasePlusCommissionEmployee
    @Override // indicates that this method overrides a superclass method
    public String toString()
    {
       return String.format(
          "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f\n%s: %.2f",
          "base-salaried commission employee", firstName, lastName,
          "social security number", socialSecurityNumber,
          "gross sales", grossSales, "commission rate", commissionRate,
          "base salary", baseSalary );
    } // end method toString
 } // end class BasePlusCommissionEmployee
```

Fig. 9.6 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 7 of 7.)

```java
// Fig. 9.7: BasePlusCommissionEmployeeTest.java
// BasePlusCommissionEmployee test program.

public class BasePlusCommissionEmployeeTest
{
   public static void main( String[] args )
   {
      // instantiate BasePlusCommissionEmployee object
      BasePlusCommissionEmployee employee =
         new BasePlusCommissionEmployee(
         "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );

      // get base-salaried commission employee data
      System.out.println(
         "Employee information obtained by get methods: \n" );
      System.out.printf( "%s %s\n", "First name is",
         employee.getFirstName() );
      System.out.printf( "%s %s\n", "Last name is",
         employee.getLastName() );
      System.out.printf( "%s %s\n", "Social security number is",
         employee.getSocialSecurityNumber() );
      System.out.printf( "%s %.2f\n", "Gross sales is",
         employee.getGrossSales() );
```

Fig. 9.7 | `BasePlusCommissionEmployee` test program. (Part 1 of 3.)

```
        System.out.printf( "%s %.2f\n", "Commission rate is",
           employee.getCommissionRate() );
        System.out.printf( "%s %.2f\n", "Base salary is",
           employee.getBaseSalary() );

        employee.setBaseSalary( 1000 ); // set base salary

        System.out.printf( "\n%s:\n\n%s\n",
           "Updated employee information obtained by toString",
            employee.toString() );
     } // end main
  } // end class BasePlusCommissionEmployeeTest
```

Fig. 9.7 | `BasePlusCommissionEmployee` test program. (Part 2 of 3.)

```
Employee information obtained by get methods:

First name is Bob
Last name is Lewis
Social security number is 333-33-3333
Gross sales is 5000.00
Commission rate is 0.04
Base salary is 300.00

Updated employee information obtained by toString:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 1000.00
```

Fig. 9.7 | `BasePlusCommissionEmployee` test program. (Part 3 of 3.)

Case Study Part 2: Creating and Using a BasePlus-CommissionEmployee Class (Cont.)

- Much of `BasePlusCommissionEmployee`'s code is similar, or identical, to that of `CommissionEmployee`.
- `private` instance variables `firstName` and `lastName` and methods `setFirstName`, `getFirstName`, `setLastName` and `getLastName` are identical.
  - Both classes also contain corresponding *get* and *set* methods.
- The constructors are almost identical
  - `BasePlusCommissionEmployee`'s constructor also sets the `base-Salary`.
- The `toString` methods are nearly identical
  - `BasePlusCommissionEmployee`'s `toString` also outputs instance variable `baseSalary`

Case Study Part 2: Creating and Using a BasePlus-CommissionEmployee Class (Cont.)

- We literally *copied* CommissionEmployee's code, pasted it into BasePlusCommissionEmployee, then modified the new class to include a base salary and methods that manipulate the base salary.
  - This "copy-and-paste" approach is often error prone and time consuming.
  - It spreads copies of the same code throughout a system, creating a **code-maintenance nightmare.**

Software Engineering Observation 9.3

With inheritance, the common instance variables and methods of all the classes in the hierarchy are declared in a superclass. When changes are made for these common features in the superclass—subclasses then inherit the changes. Without inheritance, changes would need to be made to all the source-code files that contain a copy of the code in question.

Case Study Part 3: Creating a CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy

- Class `BasePlusCommissionEmployee` class extends class `CommissionEmployee`
- A `BasePlusCommissionEmployee` object *is a* `CommissionEmployee`
  - Inheritance passes on class CommissionEmployee's capabilities.
- Class `BasePlusCommissionEmployee` also has instance variable `baseSalary`.
- Subclass `BasePlusCommissionEmployee` inherits `CommissionEmployee`'s instance variables and methods
  - Only the superclass's public and protected members are directly accessible in the subclass.

```java
// Fig. 9.8: BasePlusCommissionEmployee.java
// private superclass members cannot be accessed in a subclass.

public class BasePlusCommissionEmployee extends CommissionEmployee
{
   private double baseSalary; // base salary per week

   // six-argument constructor
   public BasePlusCommissionEmployee( String first, String last,
      String ssn, double sales, double rate, double salary )
   {
      // explicit call to superclass CommissionEmployee constructor
      super( first, last, ssn, sales, rate );

      setBaseSalary( salary ); // validate and store base salary
   } // end six-argument BasePlusCommissionEmployee constructor

```

Fig. 9.8 | `private` superclass members cannot be accessed in a subclass. (Part 1 of 5.)

```java
    // set base salary
    public void setBaseSalary( double salary )
    {
       if ( salary >= 0.0 )
          baseSalary = salary;
       else
          throw new IllegalArgumentException(
             "Base salary must be >= 0.0" );
    } // end method setBaseSalary

    // return base salary
    public double getBaseSalary()
    {
       return baseSalary;
    } // end method getBaseSalary

    // calculate earnings
    @Override // indicates that this method overrides a superclass method
    public double earnings()
    {
       // not allowed: commissionRate and grossSales private in superclass
       return baseSalary + ( commissionRate * grossSales );
    } // end method earnings
```

Fig. 9.8 | `private` superclass members cannot be accessed in a subclass. (Part 2 of 5.)

```java

    // return String representation of BasePlusCommissionEmployee
    @Override // indicates that this method overrides a superclass method
    public String toString()
    {
       // not allowed: attempts to access private superclass members
       return String.format(
          "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f\n%s: %.2f",
          "base-salaried commission employee", firstName, lastName,
          "social security number", socialSecurityNumber,
          "gross sales", grossSales, "commission rate", commissionRate,
          "base salary", baseSalary );
    } // end method toString
 } // end class BasePlusCommissionEmployee
```

Fig. 9.8 | `private` superclass members cannot be accessed in a subclass. (Part 3 of 5.)

```
BasePlusCommissionEmployee.java:39: commissionRate has private access in
CommissionEmployee
      return baseSalary + ( commissionRate * grossSales );
                           ^
BasePlusCommissionEmployee.java:39: grossSales has private access in
CommissionEmployee
      return baseSalary + ( commissionRate * grossSales );
                                             ^
BasePlusCommissionEmployee.java:49: firstName has private access in
CommissionEmployee
         "base-salaried commission employee", firstName, lastName,
                                              ^
BasePlusCommissionEmployee.java:49: lastName has private access in
CommissionEmployee
         "base-salaried commission employee", firstName, lastName,
                                                         ^
BasePlusCommissionEmployee.java:50: socialSecurityNumber has private access
in CommissionEmployee
         "social security number", socialSecurityNumber,
                                   ^
BasePlusCommissionEmployee.java:51: grossSales has private access in
CommissionEmployee
         "gross sales", grossSales, "commission rate", commissionRate,
                        ^
```

Fig. 9.8 | `private` superclass members cannot be accessed in a subclass. (Part 4 of 5.)

```
BasePlusCommissionEmployee.java:51: commissionRate has private access in
CommissionEmployee
         "gross sales", grossSales, "commission rate", commissionRate,
                                                       ^
7 errors
```

Fig. 9.8 | `private` superclass members cannot be accessed in a subclass. (Part 5 of 5.)

Case Study Part 3: Creating a CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy (Cont.)

- Each subclass constructor must implicitly or explicitly call its superclass constructor to initialize the instance variables inherited from the superclass.
  - Superclass constructor call syntax—keyword super, followed by a set of parentheses containing the superclass constructor arguments.
  - Must be the first statement in the subclass constructor's body.
- If the subclass constructor did not invoke the superclass's constructor explicitly, Java would attempt to invoke the superclass's no-argument or default constructor.
  - Class CommissionEmployee does not have such a constructor, so the compiler would issue an error.
- You can explicitly use super() to call the superclass's no-argument or default constructor, but this is rarely done.

Case Study Part 4: CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy Using protected Instance Variables

- To enable a subclass to directly access superclass instance variables, we can declare those members as protected in the superclass.
- New CommissionEmployee class modified only lines 6–10 of Fig. 9.4 as follows:

```
protected String firstName;
protected String lastName;
protected String socialSecurityNumber;
protected double grossSales;
protected double commissionRate;
```

- With protected instance variables, the subclass gets access to the instance variables, but classes that are not subclasses and classes that are not in the same package cannot access these variables directly.

```java
// Fig. 9.9: BasePlusCommissionEmployee.java
// BasePlusCommissionEmployee inherits protected instance
// variables from CommissionEmployee.

public class BasePlusCommissionEmployee extends CommissionEmployee
{
   private double baseSalary; // base salary per week

   // six-argument constructor
   public BasePlusCommissionEmployee( String first, String last,
      String ssn, double sales, double rate, double salary )
   {
      super( first, last, ssn, sales, rate );
      setBaseSalary( salary ); // validate and store base salary
   } // end six-argument BasePlusCommissionEmployee constructor

```

Fig. 9.9 | `BasePlusCommissionEmployee` inherits `protected` instance variables from `CommissionEmployee`. (Part 1 of 3.)

```java
    // set base salary
    public void setBaseSalary( double salary )
    {
       if ( salary >= 0.0 )
          baseSalary = salary;
       else
          throw new IllegalArgumentException(
             "Base salary must be >= 0.0" );
    } // end method setBaseSalary

    // return base salary
    public double getBaseSalary()
    {
       return baseSalary;
    } // end method getBaseSalary

    // calculate earnings
    @Override // indicates that this method overrides a superclass method
    public double earnings()
    {
       return baseSalary + ( commissionRate * grossSales );
    } // end method earnings
```

Fig. 9.9 | `BasePlusCommissionEmployee` inherits `protected` instance variables from `CommissionEmployee`. (Part 2 of 3.)

```java

    // return String representation of BasePlusCommissionEmployee
    @Override // indicates that this method overrides a superclass method
    public String toString()
    {
       return String.format(
          "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f\n%s: %.2f",
          "base-salaried commission employee", firstName, lastName,
          "social security number", socialSecurityNumber,
          "gross sales", grossSales, "commission rate", commissionRate,
          "base salary", baseSalary );
    } // end method toString
 } // end class BasePlusCommissionEmployee
```

Fig. 9.9 | `BasePlusCommissionEmployee` inherits `protected` instance variables from `CommissionEmployee`. (Part 3 of 3.)

Case Study Part 4: CommissionEmployee–BasePlus-CommissionEmployee Inheritance Hierarchy Using protected Instance Variables (Cont.)

- Class `BasePlusCommissionEmployee` (Fig. 9.9) extends the new version of class `CommissionEmployee` with `protected` instance variables.
  - These variables are now `protected` members of `BasePlusCommissionEmployee`.
- If another class extends this version of class `BasePlusCommissionEmployee`, the new subclass also can access the `protected` members.
- The source code in Fig. 9.9 (51 lines) is considerably shorter than that in Fig. 9.6 (128 lines)
  - Most of the functionality is now inherited from `CommissionEmployee`
  - There is now only one copy of the functionality.
  - Code is easier to maintain, modify and debug—the code related to a commission employee exists only in class `CommissionEmployee`.

Case Study Part 4: CommissionEmployee–BasePlus-CommissionEmployee Inheritance Hierarchy Using protected Instance Variables (Cont.)

- Inheriting **`protected`** instance variables slightly increases performance, because we can directly access the variables in the subclass without incurring the overhead of a *set or get method call.*

- In most cases, it's better to use **`private`** instance variables to encourage proper software engineering, and leave code optimization issues to the compiler.
  - Code will be easier to maintain, modify and debug.

Case Study Part 4: CommissionEmployee–BasePlus-CommissionEmployee Inheritance Hierarchy Using protected Instance Variables (Cont.)

- Using **`protected`** instance variables creates several potential problems.
- The subclass object can set an inherited variable's value directly without using a *set method.*
  - A subclass object can assign an invalid value to the variable
- Subclass methods are more likely to be written so that they depend on the superclass's data implementation.
  - Subclasses should depend only on the superclass services and not on the superclass data implementation.
- We may need to modify all the subclasses of the superclass if the superclass implementation changes.
  - You should be able to change the superclass implementation while still providing the same services to the subclasses.

Error-Prevention Tip 9.2

When possible, do not include `protected` *instance variables in a superclass. Instead, include non-*`private` *methods that access* `private` *instance variables. This will help ensure that objects of the class maintain consistent states.*

Case Study Part 5: CommissionEmployee–BasePlus-CommissionEmployee Inheritance Hierarchy Using private Instance Variables => BEST DESIGN

```java
// Fig. 9.10: CommissionEmployee.java
// CommissionEmployee class uses methods to manipulate its
// private instance variables.
public class CommissionEmployee
{
   private String firstName;
   private String lastName;
   private String socialSecurityNumber;
   private double grossSales; // gross weekly sales
   private double commissionRate; // commission percentage

   // five-argument constructor
   public CommissionEmployee( String first, String last, String ssn,
      double sales, double rate )
   {
      // implicit call to Object constructor occurs here
      firstName = first;
      lastName = last;
      socialSecurityNumber = ssn;
      setGrossSales( sales ); // validate and store gross sales
      setCommissionRate( rate ); // validate and store commission rate
   } // end five-argument CommissionEmployee constructor
```

instance variables are declared as private and public methods for manipulating these are provided.

Fig. 9.10 | `CommissionEmployee` class uses methods to manipulate its `private` instance variables. (Part 1 of 6.)

```

   // set first name
   public void setFirstName( String first )
   {
      firstName = first; // should validate
   } // end method setFirstName

   // return first name
   public String getFirstName()
   {
      return firstName;
   } // end method getFirstName

   // set last name
   public void setLastName( String last )
   {
      la5stName = last; // should validate
   } // end method setLastName

```

Fig. 9.10 | `CommissionEmployee` class uses methods to manipulate its `private` instance variables. (Part 2 of 6.)

```
    // return last name
    public String getLastName()
    {
       return lastName;
    } // end method getLastName

    // set social security number
    public void setSocialSecurityNumber( String ssn )
    {
       socialSecurityNumber = ssn; // should validate
    } // end method setSocialSecurityNumber

    // return social security number
    public String getSocialSecurityNumber()
    {
       return socialSecurityNumber;
    } // end method getSocialSecurityNumber

```

Fig. 9.10 | `CommissionEmployee` class uses methods to manipulate its `private` instance variables. (Part 3 of 6.)

```
   // set gross sales amount
   public void setGrossSales( double sales )
   {
      if ( sales >= 0.0 )
         grossSales = sales;
      else
         throw new IllegalArgumentException(
            "Gross sales must be >= 0.0" );
   } // end method setGrossSales

   // return gross sales amount
   public double getGrossSales()
   {
      return grossSales;
   } // end method getGrossSales

```

Fig. 9.10 | `CommissionEmployee` class uses methods to manipulate its `private` instance variables. (Part 4 of 6.)

```java
    // set commission rate
    public void setCommissionRate( double rate )
    {
       if ( rate > 0.0 && rate < 1.0 )
          commissionRate = rate;
       else
          throw new IllegalArgumentException(
             "Commission rate must be > 0.0 and < 1.0" );
    } // end method setCommissionRate

    // return commission rate
    public double getCommissionRate()
    {
       return commissionRate;
    } // end method getCommissionRate

    // calculate earnings
    public double earnings()
    {
       return getCommissionRate() * getGrossSales();
    } // end method earnings

```

Fig. 9.10 | `CommissionEmployee` class uses methods to manipulate its `private` instance variables. (Part 5 of 6.)

```java
    // return String representation of CommissionEmployee object
    @Override // indicates that this method overrides a superclass method
    public String toString()
    {
       return String.format( "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",
          "commission employee", getFirstName(), getLastName(),
          "social security number", getSocialSecurityNumber(),
          "gross sales", getGrossSales(),
          "commission rate", getCommissionRate() );
    } // end method toString
 } // end class CommissionEmployee
```

Fig. 9.10 | `CommissionEmployee` class uses methods to manipulate its `private` instance variables. (Part 6 of 6.)

Case Study Part 5: CommissionEmployee–BasePlus-CommissionEmployee Inheritance Hierarchy Using private Instance Variables (Cont.)

- `CommissionEmployee` methods `earnings` and `toString` use the class's *get* methods to obtain the values of its instance variables.
  - If we decide to change the internal representation of the data (e.g., variable names) only the bodies of the *get and set methods that directly manipulate the instance variables will need to change.*
  - These changes occur solely within the superclass-—no changes to the subclass are needed.
  - Localizing the effects of changes like this is a good software engineering practice.
- Subclass `BasePlusCommissionEmployee` inherits `CommissionEmployee's` non-private methods and can access the private superclass members via those methods.

Case Study Part 5: CommissionEmployee–BasePlus-CommissionEmployee Inheritance Hierarchy Using private Instance Variables (Cont.)

```java
// Fig. 9.11: BasePlusCommissionEmployee.java
// BasePlusCommissionEmployee class inherits from CommissionEmployee
// and accesses the superclass's private data via inherited
// public methods.

public class BasePlusCommissionEmployee extends CommissionEmployee
{
   private double baseSalary; // base salary per week

   // six-argument constructor
   public BasePlusCommissionEmployee( String first, String last,
      String ssn, double sales, double rate, double salary )
   {
      super( first, last, ssn, sales, rate );
      setBaseSalary( salary ); // validate and store base salary
   } // end six-argument BasePlusCommissionEmployee constructor

```

Fig. 9.11 | `BasePlusCommissionEmployee` class inherits from `CommissionEmployee` and accesses the superclass's `private` data via inherited `public` methods. (Part 1 of 3.)

```java
    // set base salary
    public void setBaseSalary( double salary )
    {
       if ( salary >= 0.0 )
          baseSalary = salary;
       else
          throw new IllegalArgumentException(
             "Base salary must be >= 0.0" );
    } // end method setBaseSalary

    // return base salary
    public double getBaseSalary()
    {
       return baseSalary;
    } // end method getBaseSalary

    // calculate earnings
    @Override // indicates that this method overrides a superclass method
    public double earnings()
    {
       return getBaseSalary() + super.earnings();
    } // end method earnings
```

Method `earnings` overrides class the superclass's `earnings` method.

calls `CommissionEmployee's earnings` method with `super.earnings()`

Good software engineering practice: If a method performs all or some of the actions needed by another method, call that method rather than duplicate its code.

```java

    // return String representation of BasePlusCommissionEmployee
    @Override // indicates that this method overrides a superclass method
    public String toString()
    {
       return String.format( "%s %s\n%s: %.2f", "base-salaried",
          super.toString(), "base salary", getBaseSalary() );
    } // end method toString
 } // end class BasePlusCommissionEmployee
```

Fig. 9.11 | `BasePlusCommissionEmployee` class inherits from `CommissionEmployee` and accesses the superclass's `private` data via inherited `public` methods. (Part 3 of 3.)

BasePlusCommissionEmployee's toString method overrides class CommissionEmployee's toString method

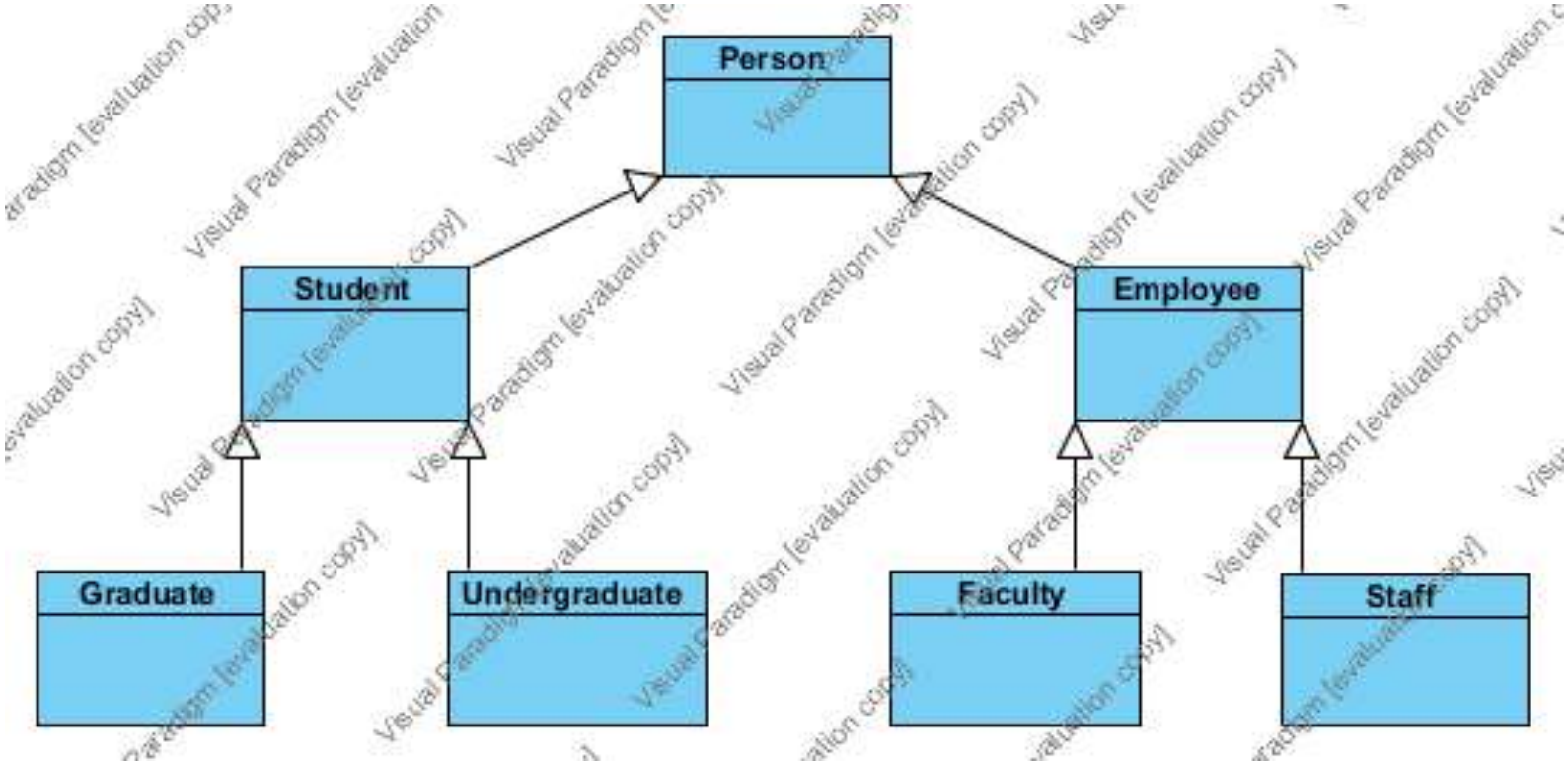
The new version creates part of the String representation by calling CommissionEmployee's toString method with the expression super.toString().

Constructors in Subclasses

- Instantiating a subclass object begins a chain of constructor calls
  - The subclass constructor, before performing its own tasks, invokes its direct superclass's constructor
- If the superclass is derived from another class, the superclass constructor invokes the constructor of the next class up the hierarchy, and so on.
- The last constructor called in the chain is always class Object's constructor.
- Original subclass constructor's body finishes executing last.
- Each superclass's constructor manipulates the superclass instance variables that the subclass object inherits.

UML Inheritance Diagrams

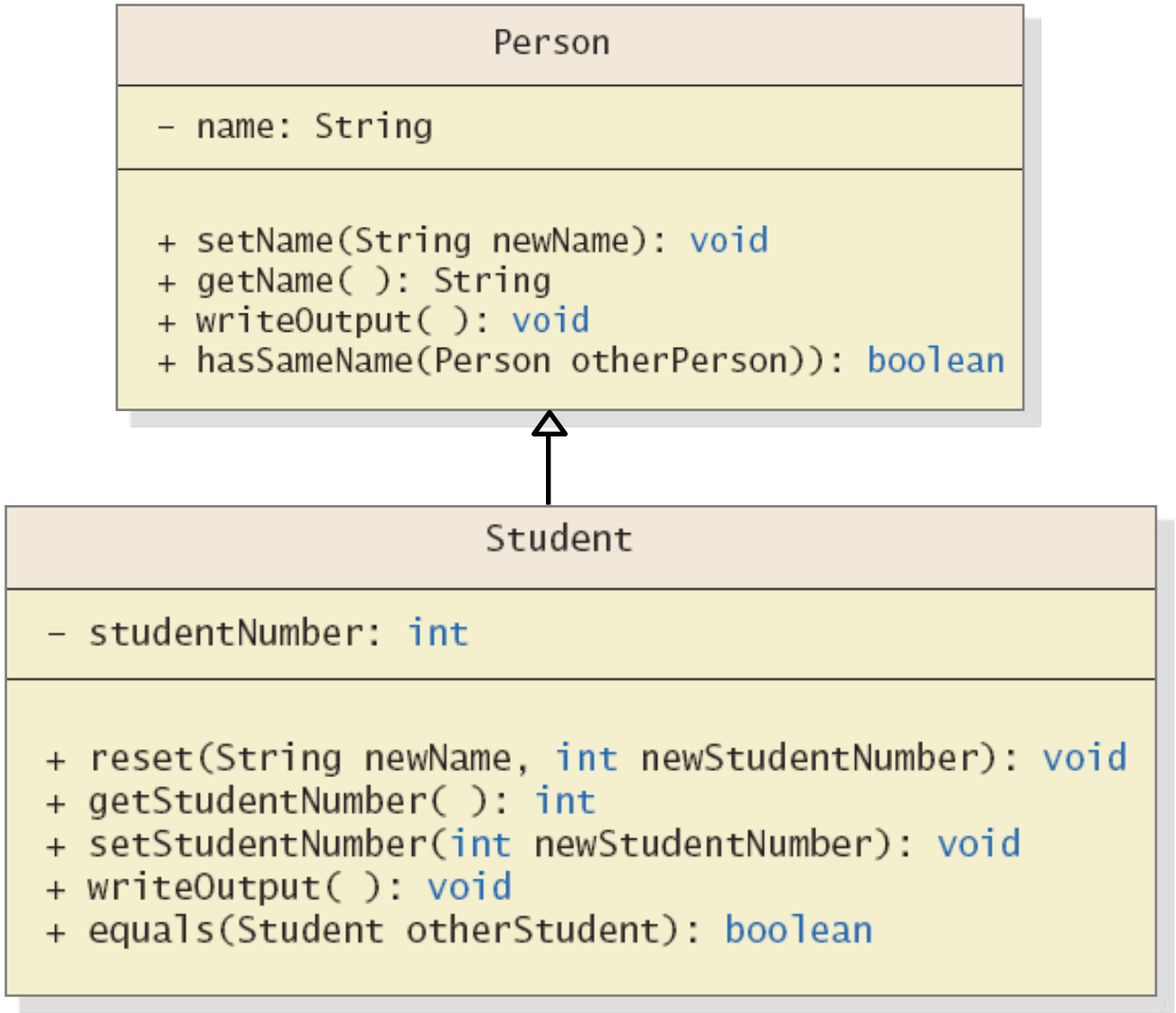
A class hierarchy in UML notation



An Employee is a Person and so forth; hence the arrows point up.

UML Inheritance Diagrams

- Some details of UML class hierarchy from previous figure



Acknowledgments

- The course material used to prepare this presentation is mostly taken/adopted from the list below:
  - Java - How to Program, Paul Deitel and Harvey Deitel, Prentice Hall, 2012
  - Java - An Introduction to Problem Solving and Programming, Walter Savitch, Pearson, 2012

BBM 102 – Introduction to Programming II

Spring 2017

Polymorphism

Instructors: Ayça Tarhan, Fuat Akal, Gönenç Ercan, Vahid Garousi
TAs: Selma Dilek, Selim Yılmaz, Selman Bozkır

Today

- Inheritance revisited
- Comparing objects : `equals()` method
- `instanceof` keyword
- Polymorphism

Visibility Revisited

- All variables and methods of a parent class, even private members, are inherited by its children

- As we've mentioned, private members cannot be referenced by name in the child class

- However, private members inherited by child classes exist and can be referenced indirectly
  - Because the parent can refer to the private member, the child can reference it indirectly using its parent's methods
  - The `super` reference can be used to refer to the parent class, even if no object of the parent class exists

Parent-child class relationship

- An example of **`super`** reference

```
public class Superclass {

    public void printMethod() {
        System.out.println("Printed in Superclass.");
    }
}



public class Subclass extends Superclass {

    // overrides printMethod in Superclass
    public void printMethod() {
        super.printMethod();
        System.out.println("Printed in Subclass");
    }
    public static void main(String[] args) {
        Subclass s = new Subclass();
        s.printMethod();
    }
}
```

Output:

```
Printed in Superclass.
Printed in Subclass
```

Designing for Inheritance

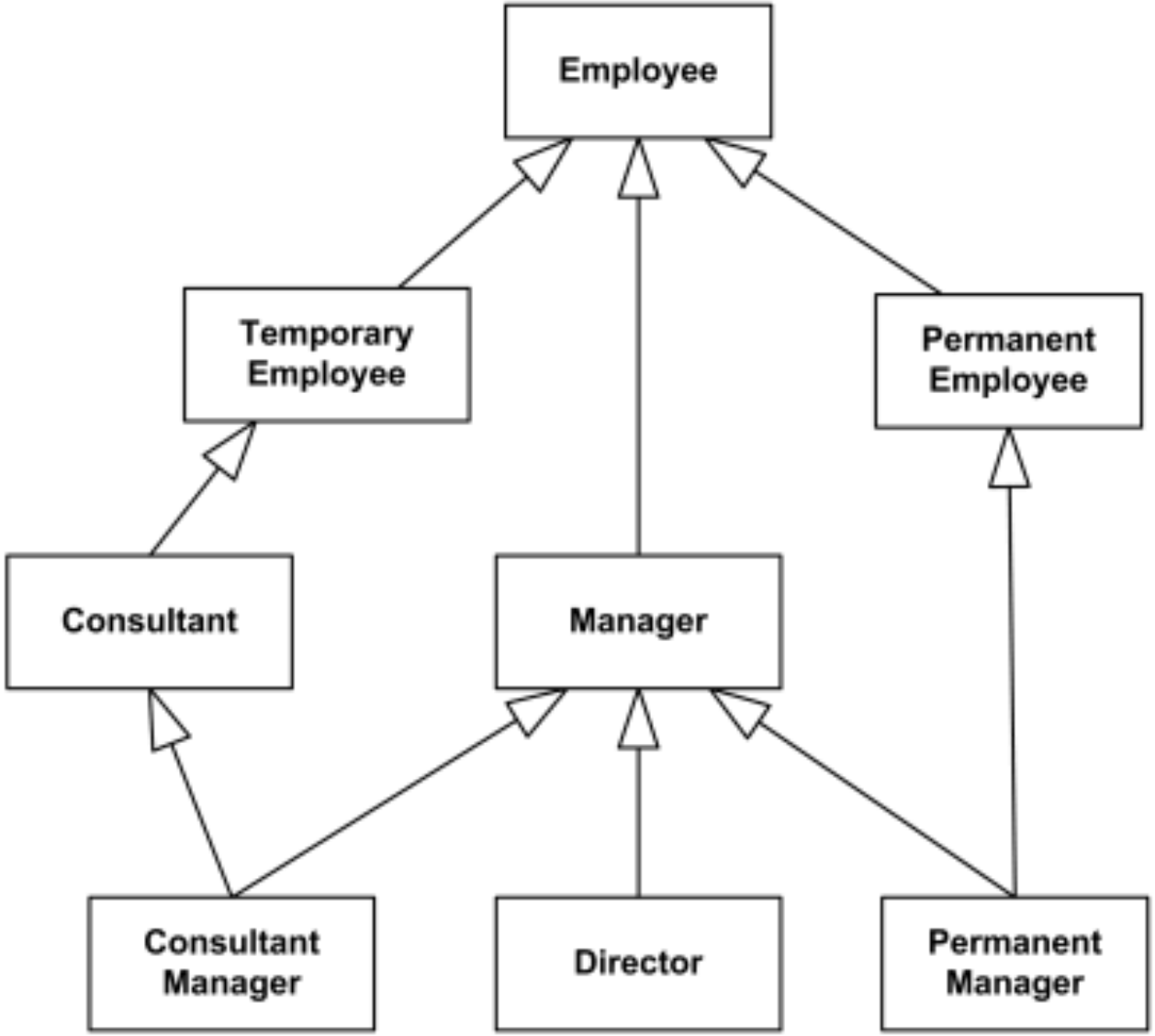
- Taking the time to create a good software design can ensure long-term benefits
- Inheritance issues are an important part of an object-oriented design
- Properly designed inheritance relationships can contribute greatly to the elegance, maintainability, and reuse of the software
- Let's summarize some of the issues regarding inheritance that relate to a good software design

Inheritance Design Issues

- Every derivation from the main class should be an is-a relationship
- Think about a potential future class hierarchy
- Design classes to be reusable and flexible
- Find common characteristics of classes and push them as high in the class hierarchy as appropriate, i.e. "generalize" the behavior
- Override methods as appropriate to tailor or change the functionality of a child
- Add new variables to children, but don't redefine (shadow) inherited variables

Inheritance Design Issues

- An example class hierarchy



More generalized

More specialized

Inheritance Design Issues

- Allow each class to manage its own data; use the `super` reference to invoke the parent's constructor to set up its data
- Even if there are no current uses for them, override general methods such as `toString` and `equals` with appropriate definitions
- Use abstract classes to represent general concepts that lower classes have in common
- Use visibility modifiers carefully to provide needed access without violating encapsulation

Restricting Inheritance

- The `final` modifier can be used to cut down inheritance
  - If the `final` modifier is applied to a method, then <u>that method cannot be overridden in any descendent classes</u>
  - If the `final` modifier is applied to an entire class, then <u>that class cannot be used to derive any children at all</u>
- These are key design decisions and establish that a method or class must be used "as is" or not at all

Restricting Inheritance

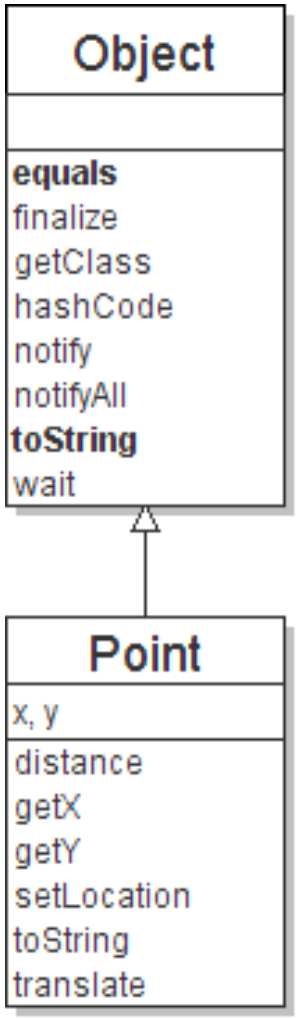
- Example of the `final` modifier

```
public class Base
{
    public       void m1() {...}
    public final void m2() {...}
}
public class Derived extends Base
{
    public void m1() {...}  // OK, overriding Base#m1()
    public void m2() {...}  // forbidden
}
```

Class Object

- In Java, all types of objects have a superclass named `Object`.
  - Every class implicitly `extends Object`.

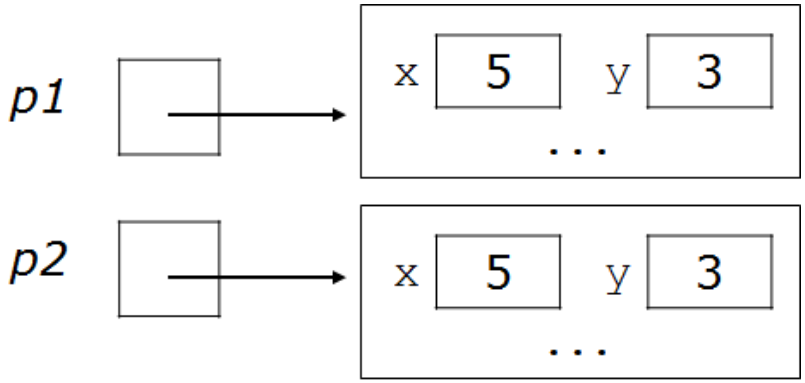
- The Object class defines several methods:
  - `public String toString()`
    Used to print the object.
  - `public boolean equals(Object other)`
    Compare the object to any other for equality.



Comparing objects

- The == operator does not work well with objects.
  - == compares references to objects, not their contents or state.
  - Example:

```
Point p1 = new Point(5, 3);
Point p2 = new Point(5, 3);
if (p1 == p2) {    // false
    System.out.println("equal");
}
```



The `equals()` method

- The `equals` method compares <u>the contents / state of objects</u>.
  - `equals` should be used when comparing `String`s, `Point`s, …

```
if (str1.equals(str2)) {
    System.out.println("the strings are equal");
}
```

- If you write your own class, its `equals` method will behave just like the == operator.

```
Point p1 = new Point(5, 3);
Point p2 = new Point(5, 3);
if (p1.equals(p2)) {    // false
    System.out.println("equal");
}
```

  - This is the behavior we inherit from class `Object`.

Initial flawed `equals()` method

- We can change this behavior by writing an `equals` method.
  - Ours will *override* the default behavior from class Object.
  - The method should compare the state of the two objects and return `true` for cases like the above.

- A flawed implementation of the `equals` method:

```
public boolean equals(Point other) {
    if (x == other.x && y == other.y) {
        return true;
    } else {
        return false;
    }
}
```

Flaws in `equals()` method

- It should be legal to compare a `Point` to any object (not just other `Point` objects):

```
// this should be allowed
Point p = new Point(7, 2);
if (p.equals("hello")) {   // false
    ...
```

  - `equals` should always return false if a non-`Point` is passed.

`equals()` and the Object class

- `equals()` method, general syntax:

```
public boolean equals(Object <name>)  {
        <statement(s) that return a boolean value> ;
}
```

  - The parameter to `equals` must be of type `Object`.
  - `Object` is a general type that can match any object.
  - Having an `Object` parameter means *any* object can be passed.

Another flawed version

- Another flawed `equals` implementation:

```
public boolean equals(Object o) {
    return x == o.x && y == o.y;
}
```

- It does not compile:

```
Point.java:36: cannot find symbol
symbol  : variable x
location: class java.lang.Object
return x == o.x && y == o.y;
            ^
```

  - The compiler is saying,
    "`o` could be any object. Not every object has an `x` field."

Type-casting objects

- Solution: *Type-cast* the object parameter to a `Point`.

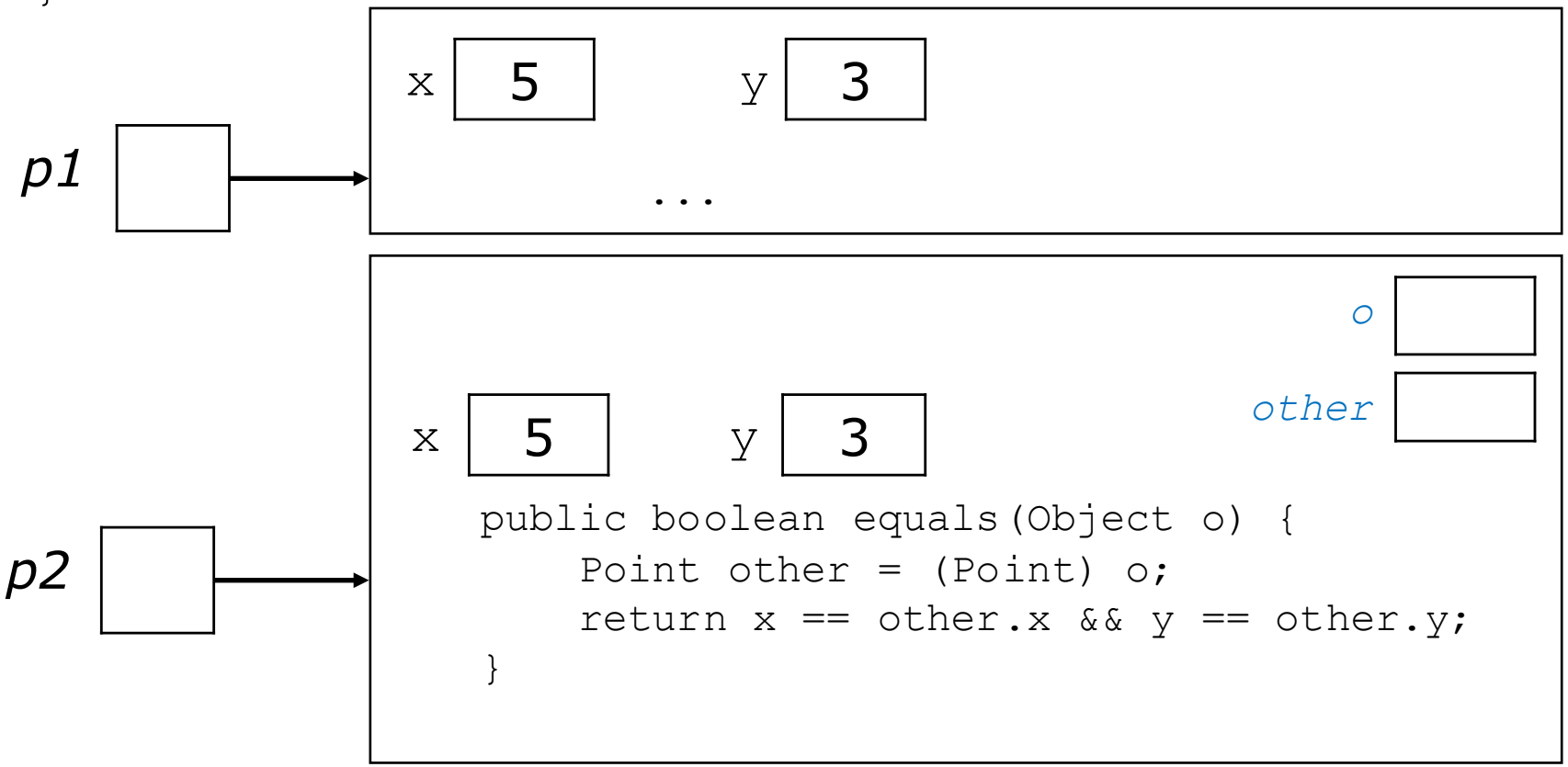
```
public boolean equals(Object o) {
    Point other = (Point) o;
    return x == other.x && y == other.y;
}
```

- Casting objects is different than casting primitives.
  - We're really casting an `Object` reference into a `Point` reference.
  - We're promising the compiler that `o` refers to a `Point` object.

Casting objects diagram

- Client code:

```
Point p1 = new Point(5, 3);
Point p2 = new Point(5, 3);
if (p1.equals(p2)) {
    System.out.println("equal");
}
```



Comparing different types

- When we compare `Point` objects to other types:

```
Point p = new Point(7, 2);
if (p.equals("hello")) {    // should be false
    ...
}
```

  - Currently the code crashes:

```
Exception in thread "main"
java.lang.ClassCastException: java.lang.String
        at Point.equals(Point.java:25)
        at PointMain.main(PointMain.java:25)
```

  - The culprit is the line with the type-cast:

```
public boolean equals(Object o) {
    Point other = (Point) o;
```

The `instanceof` keyword

- We can use a keyword called `instanceof` to ask whether a variable refers to an object of a given type.

- The `instanceof` keyword, general syntax:

  <variable> `instanceof` ***<type>***

  - The above is a `boolean` expression.

  - Example:

```
String s = "hello";
Point p = new Point();
```

expression	result
`s instanceof Point`	`false`
`s instanceof String`	`true`
`p instanceof Point`	`true`
`p instanceof String`	`false`
`null instanceof String`	`false`

Final version of `equals` method

```
// Returns whether o refers to a Point object with
// the same (x, y) coordinates as this Point object.
public boolean equals(Object o) {
    if (o instanceof Point) {
        // o is a Point; cast and compare it
        Point other = (Point) o;
        return x == other.x && y == other.y;
    } else {
        // o is not a Point; cannot be equal
        return false;
    }
}
```

- This version correctly compares `Points` to any type of object.

Polymorphism

Polymorphism

- Polymorphism means *many* (poly) *shapes* (morph) : "having many forms"

- Enables you to “program in the general” rather than “program in the specific.”

- Polymorphism enables you to write programs that process objects that share the same superclass as if they’re all objects of the superclass; this can simplify programming.

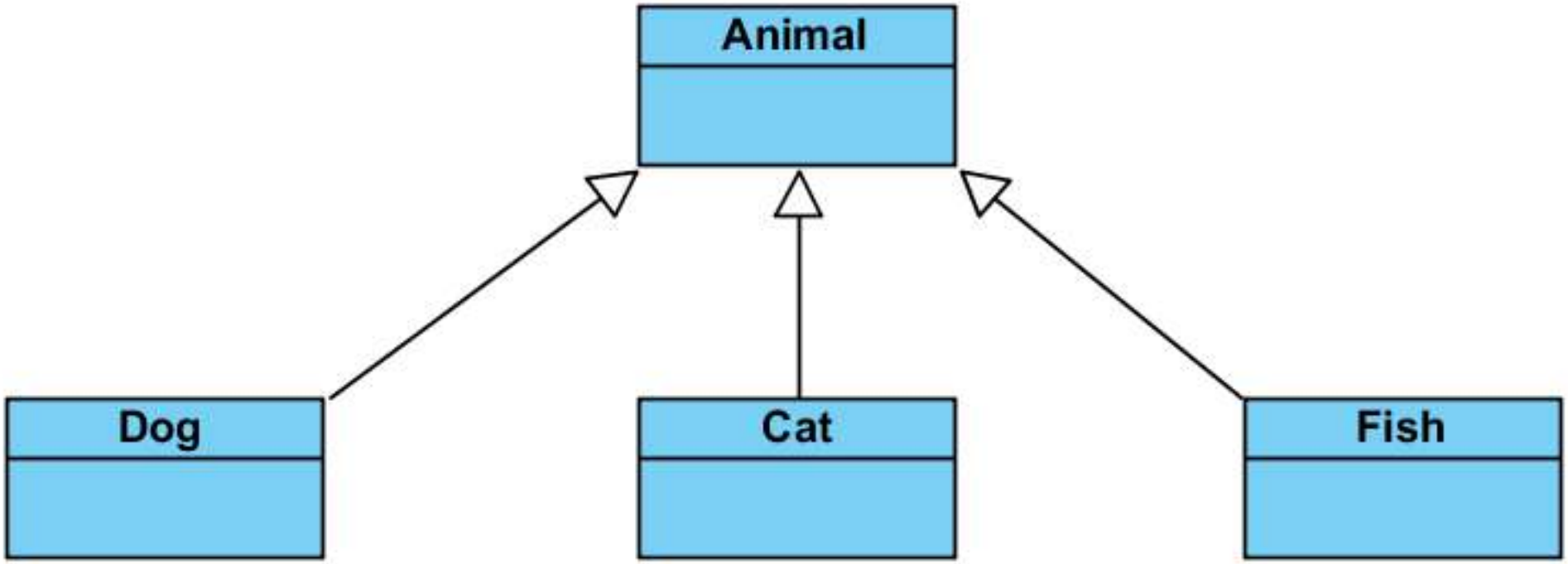
Polymorphism

- A *polymorphic reference* is a variable that can refer to different types of objects at different points in time

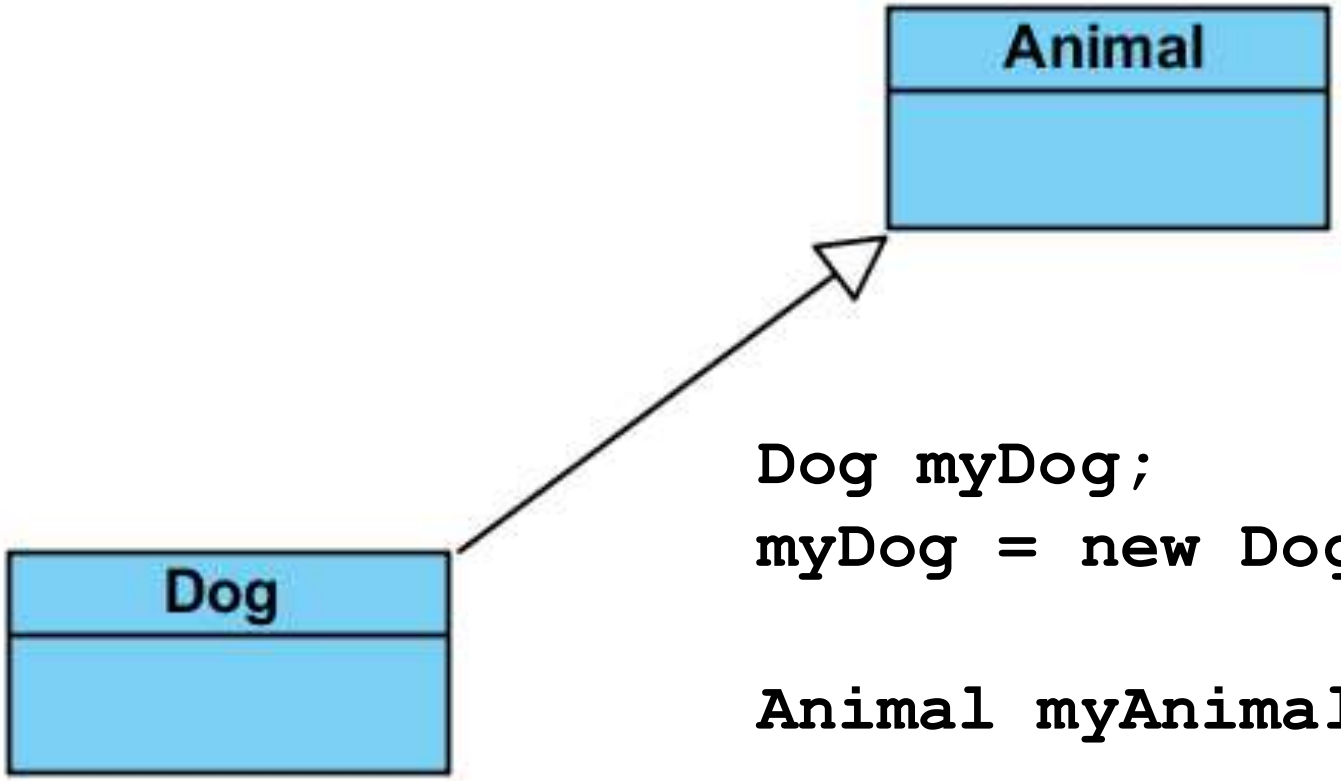
- All object references in Java are potentially polymorphic and can refer to an object of any type compatible with its defined type

- Compatibility of class types can be based on either Inheritance or Interfaces (which we will see later)

An Example Class Hierarchy



A Polymorphic Example



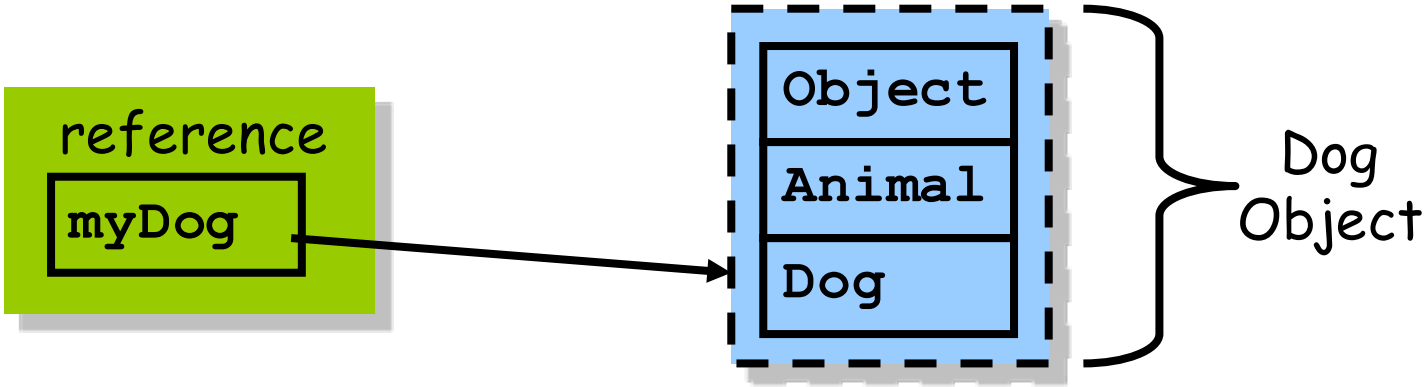

```
Dog myDog;
myDog = new Dog();

Animal myAnimal;

myAnimal = myDog;
```

Everything is an Object!

- When we say:

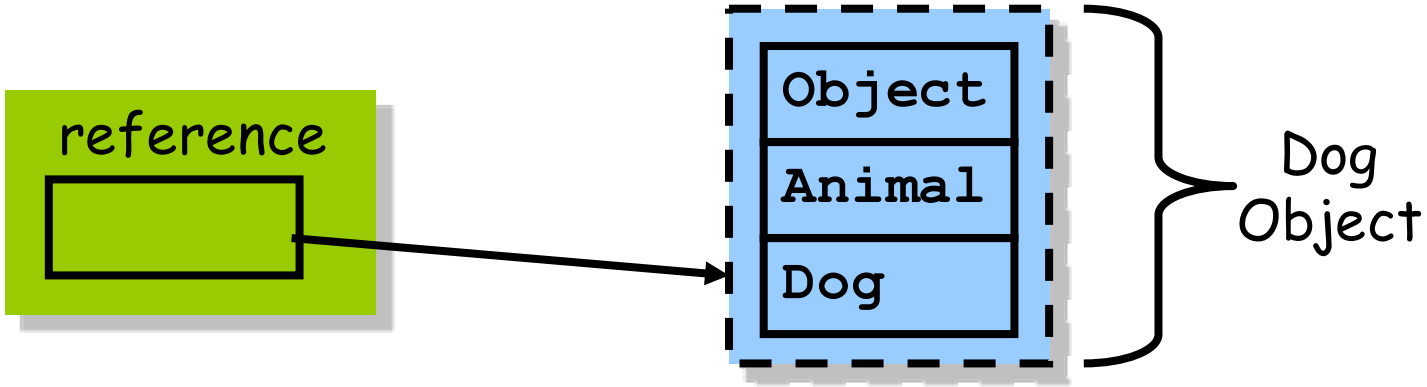
  myDog = new Dog();

- the Dog constructor gets called.
- It, in turn, must call the Animal constructor
- When you don't extend anything, by default you extend Object
- Thus the Animal constructor calls the Object constructor
- Looking at an object in memory it will look like something like this:



Polymorphism Explained

- The rule is very simple
- A reference can refer to an object which is either
  - The same type as the reference
  - Has a superclass of the same type as the reference
- So all of the following are legal
  - Dog d = new Dog();
  - Animal a = new Animal();
  - Object o = new Object();



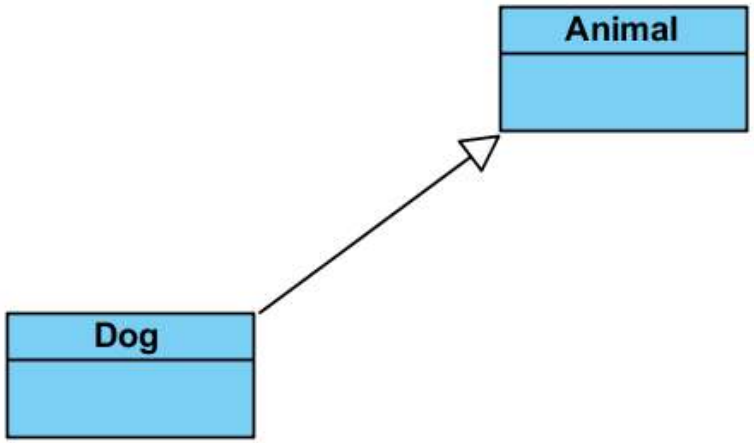
An Illegal Example

- We are able to assign an object of a sub-class into an object of a super-class as in:

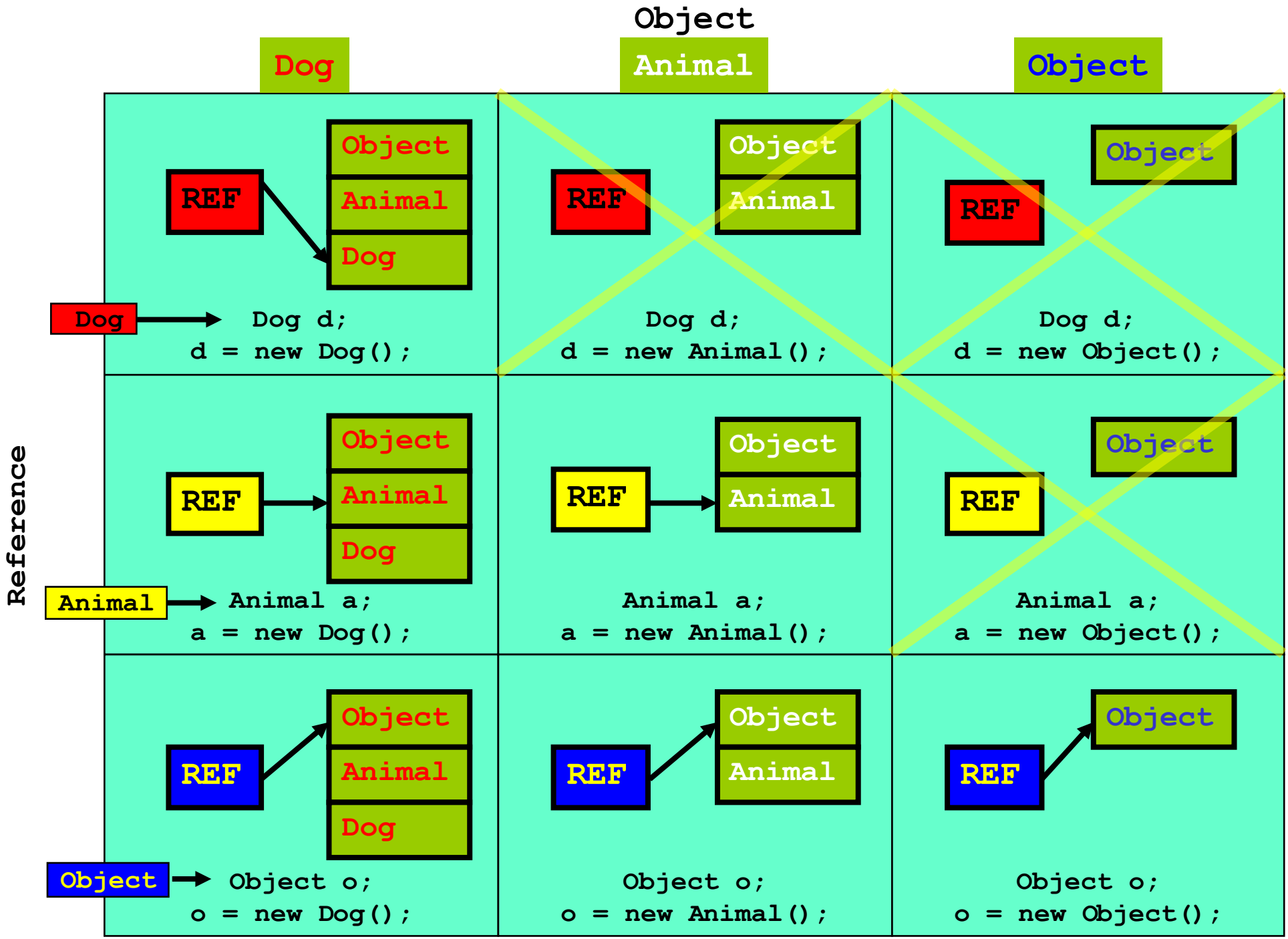
```
Animal MyAnimal = new Dog();
```

- But the reverse is not true. We can't assign a superclass object into a sub-class object.

```
Dog MyDog = new Animal(); // illegal
```



All dogs are animals but not all animals are dogs


Object
Dog
Animal
Object
Reference
REF
Object
Animal
Dog
Dog
Dog d;
d = new Dog();
REF
Object
Animal
Dog d;
d = new Animal();
REF
Object
Dog d;
d = new Object();
REF
Object
Animal
Dog
Animal
Animal a;
a = new Dog();
REF
Object
Animal
Animal a;
a = new Animal();
REF
Object
Animal a;
a = new Object();
REF
Object
Animal
Dog
Object
Object o;
o = new Dog();
REF
Object
Animal
Object o;
o = new Animal();
REF
Object
Object o;
o = new Object();

Polymorphism Examples

- **Example:** Suppose we create a program that simulates the movement of several types of animals for a biological study. Classes Fish, Frog and Bird represent the three types of animals under investigation.
  - Each class extends superclass Animal, which contains a method move and maintains an animal's current location as x-y coordinates. Each subclass implements method move.
  - A program maintains an Animal array containing references to objects of the various Animal subclasses. To simulate the animals' movements, the program sends each object the same message once per second—namely, move.

Polymorphism Examples

- Each specific type of Animal responds to a move message in a unique way:
  - a Fish might swim three meters
  - a Frog might jump five meters
  - a Bird might fly ten meters.

- The program issues the same message (i.e., move) to each animal object, but each object knows how to modify its x-y coordinates appropriately for its specific type of movement.
- Relying on each object to know how to "do the right thing" in response to the same method call is the key concept of polymorphism.
- The same message sent to a variety of objects has "many forms" of results—hence the term *polymorphism*.

Polymorphism Examples (Cont.)

- **Example**: Space Objects in a Video Game
  - A video game manipulates objects of classes Martian, Venusian, Plutonian, SpaceShip and LaserBeam. Each inherits from SpaceObject and overrides its draw method.
  - A screen manager maintains a collection of references to objects of the various classes and periodically sends each object the same message—namely, draw.
  - Each object responds in a unique way.
    - A Martian object might draw itself in red with green eyes and the appropriate number of antennae.
    - A SpaceShip object might draw itself as a bright silver flying saucer.
    - A LaserBeam object might draw itself as a bright red beam across the screen.

The same message (in this case, draw) sent to a variety of objects has "many forms" of results.

Polymorphism Examples (Cont.)

- A screen manager might use polymorphism to facilitate adding new classes to a system with minimal modifications to the system's code.

- To add new objects to our video game:
  - Build a class that extends SpaceObject and provides its own draw method implementation.
  - When objects of that class appear in the SpaceObject collection, the screen manager code invokes method draw, exactly as it does for every other object in the collection, regardless of its type.
  - So the new objects simply "plug right in" without any modification of the screen manager code by the programmer.

Demonstrating Polymorphic Behavior

- A superclass object cannot be treated as a subclass object, because a superclass object is *not* an object of any of its subclasses.
- The *is-a* relationship applies only up the hierarchy from a subclass to its direct (and indirect) superclasses, and not down the hierarchy.
- The Java compiler *does* allow the assignment of a superclass reference to a subclass variable if you explicitly cast the superclass reference to the subclass type
  - A technique known as downcasting that enables a program to invoke subclass methods that are not in the superclass.

Demonstrating Polymorphic Behavior (Cont.)

- When a superclass variable contains a reference to a subclass object, and that reference is used to call a method, the subclass version of the method is called.
  - The Java compiler allows this "crossover" because an object of a subclass *is an object of its superclass (but not vice versa).*
- When the compiler encounters a method call made through a variable, the compiler determines if the method can be called by checking the variable's class type.
  - If that class contains the proper method declaration (or inherits one), the call is compiled.
- At execution time, the type of the object to which the variable refers determines the actual method to use.
  - This process is called *dynamic binding*.

Method Calls and Polymorphism

Assume the Dog class extends the Animal class, redefining the "makeNoise" method.

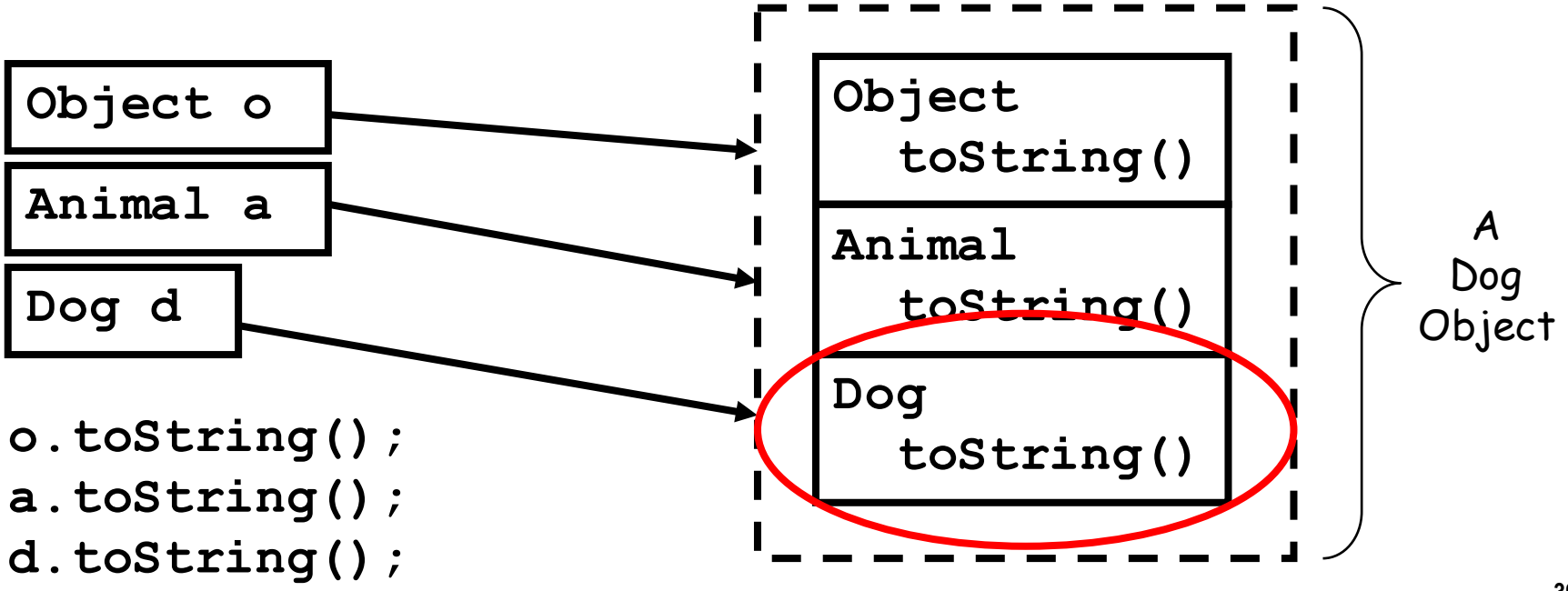
Consider the following:

```
Animal myAnimal = new Dog();
myAnimal.makeNoise();
```

Note: The Animal reference is referring to a Dog object. And it is the Dog's makeNoise method that gets invoked!

Dynamic Binding

- Very simple rule.
  - No matter what the reference type is, Java will search the object and execute the lowest occurrence of a method it finds.
- class Object has a toString method
- Assume that both Animal and Dog have overridden the toString method



Polymorphism

- With polymorphism, we can design and implement systems that are easily extensible

- New classes can be added with little or no modification to the general portions of the program, as long as the new classes are part of the inheritance hierarchy.

- The only parts of a program that must be altered for new classes are those that require direct knowledge of the new classes.

Polymorphism

- A variable of a type T can legally refer to an object of any subclass of T.

```
Employee person = new Lawyer();
System.out.println(person.getSalary());        // 50000.0
System.out.println(person.getVacationForm());  // pink
```

- You can call any methods from `Employee` on the variable `person`, but not any methods specific to `Lawyer` (such as `Sue`).

- Once a method is called on the object, it behaves in its normal way (as a `Lawyer`, not as a normal `Employee`).

Polymorphism + parameters

- You can declare methods to accept superclass types as parameters, then pass a parameter of any subtype.

```
public class EmployeeMain {
        public static void main(String[] args) {
            Lawyer lisa = new Lawyer();
            Secretary steve = new Secretary();
            printInfo(lisa);
            printInfo(steve);
        }

        public static void printInfo(Employee empl) {
            System.out.println("salary = " + empl.getSalary());
            System.out.println("days = " + empl.getVacationDays());
            System.out.println("form = " + empl.getVacationForm());
            System.out.println();
        }
    }
```

OUTPUT:

```
salary = 50000.0
vacation days = 21
vacation form = pink

salary = 50000.0
vacation days = 10
vacation form = yellow
```

- You can declare arrays of superclass types, and store objects of any subtype as elements.

```
public class EmployeeMain2 {
        public static void main(String[] args) {
            Employee[] employees = {new Lawyer(), new Secretary(),
                    new Marketer(), new LegalSecretary()};

            for (int i = 0; i < employees.length; i++) {
                System.out.println("salary = " +
                                    employees[i].getSalary());
                System.out.println("vacation days = " +
                                    employees[i].getVacationDays());
                System.out.println();
            }
        }
    }
```

OUTPUT:

```
salary = 50000.0
vacation days = 15

salary = 50000.0
vacation days = 10

salary = 60000.0
vacation days = 10

salary = 55000.0
vacation days = 10
```

Polymorphism vs. Inheritance

- Inheritance is required in order to achieve polymorphism (we must have class hierarchies).
  - Re-using class definitions via extension and redefinition

- Polymorphism is not required in order to achieve inheritance.
  - An object of class A acts as an object of class B (an ancestor to A).

References and Inheritance

- Assigning a child object to a parent reference is considered to be a *widening conversion*, and can be performed by simple assignment
  - The widening conversion is the most useful

- Assigning a parent object to a child reference can be done, but it is considered a *narrowing conversion* and two rules/guidelines apply:
  - A narrowing conversion must be done with a cast
  - A narrowing conversion should only be <u>used to restore an object back to its original class</u> (back to what it was "born as" with the new operator)

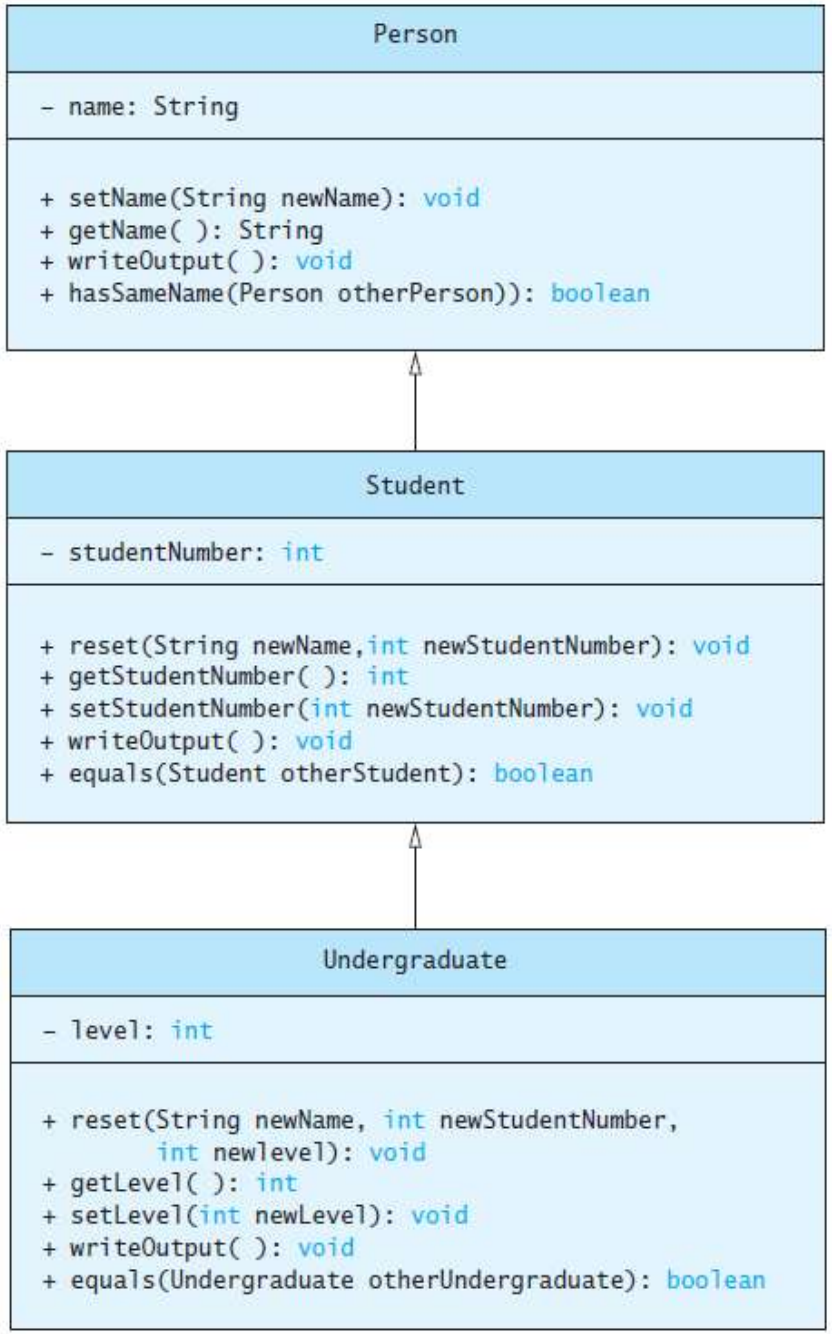
Polymorphism Example

- Consider an array of **Person**

```
Person[] people = new Person[4];
```

- Since **Student** and **Undergraduate** are types of **Person**, we can assign them to **Person** variables

```
people[0] = new Student("DeBanque, Robin", 8812);

people[1] = new Undergraduate("Cotty, Manny", 8812, 1);
```



Example

- Given:

```
Person[] people = new Person[4];
people[0] = new Student("DeBanque, Robin", 8812);
```

- When invoking:

```
people[0].writeOutput();
```

- Which `writeOutput()` is invoked, the one defined for `Student` or the one defined for `Person`?

- Answer: The one defined for `Student`

Example

```
public class PolymorphismDemo
{
    public static void main(String[] args) {
        Person[] people = new Person[4];
        people[0] = new Undergraduate("Cotty, Manny", 4910, 1);
        people[1] = new Undergraduate("Kick, Anita", 9931, 2);
        people[2] = new Student("DeBanque, Robin", 8812);
        people[3] = new Undergraduate("Bugg, June", 9901, 4);

        for (Person p : people)
        {
            p.writeOutput();
            System.out.println();
        }
    }
}
```

```
Name: Cotty, Manny
Student Number: 4910
Student Level: 1

Name: Kick, Anita
Student Number: 9931
Student Level: 2

Name: DeBanque, Robin
Student Number: 8812

Name: Bugg, June
Student Number: 9901
Student Level: 4
```

A polymorphism problem

- Assume that the following four classes have been declared:

```
public class Foo {
    public void method1() {
        System.out.println("foo 1");
    }

    public void method2() {
        System.out.println("foo 2");
    }

    public String toString() {
        return "foo";
    }
}
```

(continued on next slide)

A polymorphism problem (cont'd)

```
public class Bar extends Foo {
    public void method2() {
        System.out.println("bar 2");
    }
}

public class Baz extends Foo {
    public void method1() {
        System.out.println("baz 1");
    }

    public String toString() {
        return "baz";
    }
}

public class Mumble extends Baz {
    public void method2() {
        System.out.println("mumble 2");
    }
}
```

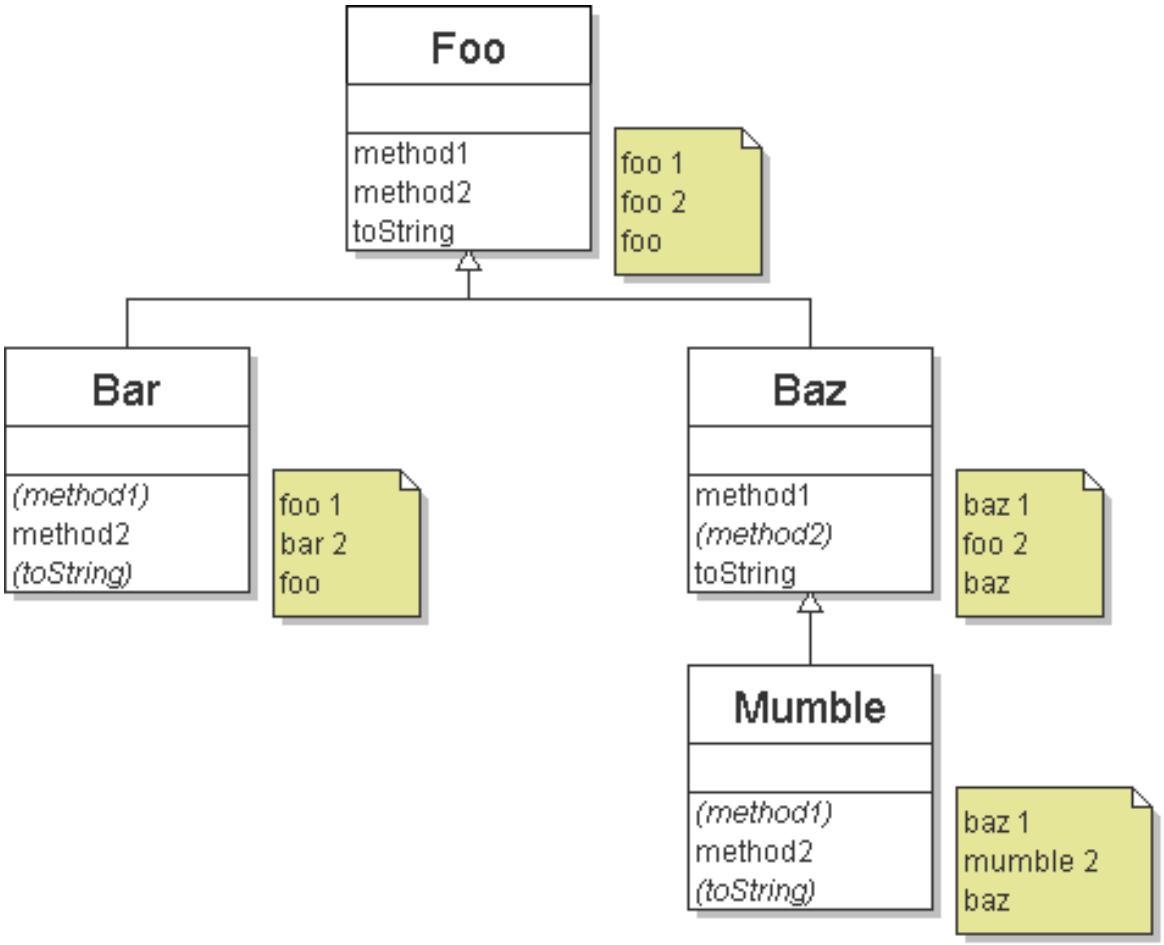
A polymorphism problem (cont'd)

- What would be the output of the following client code?

```
Foo[] pity = {new Baz(), new Bar(), new Mumble(), new Foo()};
for (int i = 0; i < pity.length; i++) {
    System.out.println(pity[i]);
    pity[i].method1();
    pity[i].method2();
    System.out.println();
}
```

Finding output with diagrams

- One way to determine the output is to diagram each class and its methods, including their output:
  - Add the classes from top (superclass) to bottom (subclass).
  - Include any inherited methods and their output.



Finding output with tables

- Another possible technique for solving these problems is to make a table of the classes and methods, writing the output in each square.

method	Foo	Bar	Baz	Mumble
`method1`	`foo 1`	*`foo 1`*	`baz 1`	*`baz 1`*
`method2`	`foo 2`	`bar 2`	*`foo 2`*	`mumble 2`
`toString`	`foo`	*`foo`*	`baz`	*`baz`*

Polymorphism answer

```
Foo[] pity = {new Baz(), new Bar(), new Mumble(), new Foo()};
for (int i = 0; i < pity.length; i++) {
    System.out.println(pity[i]);
    pity[i].method1();
    pity[i].method2();
    System.out.println();
}
```

Output:

```
baz
baz 1
foo 2

foo
foo 1
bar 2

baz
baz 1
mumble 2

foo
foo 1
foo 2
```

Another problem

- Assume that the following classes have been declared:
  - The order of classes is changed, as well as the client.
  - The methods now sometimes call other methods.

```
public class Lamb extends Ham {
    public void b() {
        System.out.print("Lamb b   ");
    }
}
public class Ham {
    public void a() {
        System.out.print("Ham a   ");
        b();
    }
    public void b() {
        System.out.print("Ham b   ");
    }
    public String toString() {
        return "Ham";
    }
}
```

(continued on next slide)

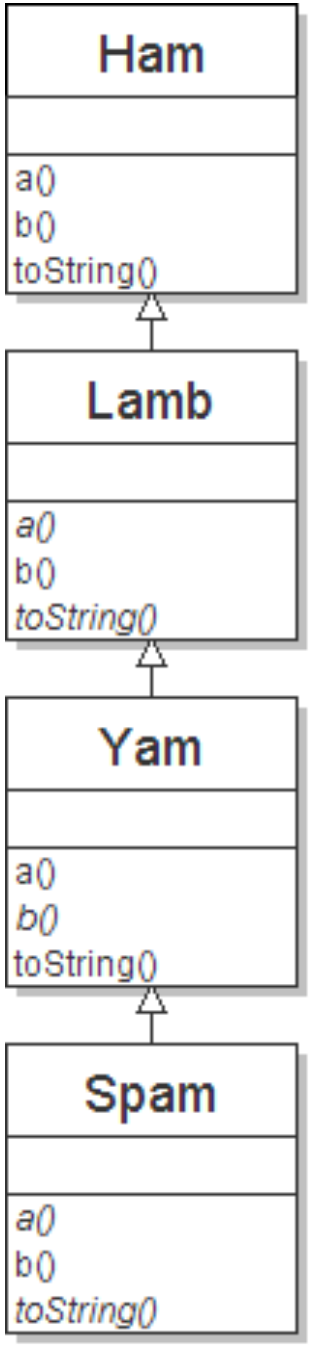
```
public class Spam extends Yam {
    public void b() {
        System.out.print("Spam b   ");
    }
}

public class Yam extends Lamb {
    public void a() {
        System.out.print("Yam a   ");
        super.a();
    }

    public String toString() {
        return "Yam";
    }
}
```

- What would be the output of the following client code?

```
Ham[] food = {new Spam(), new Yam(), new Ham(), new Lamb()};
for (int i = 0; i < food.length; i++) {
    System.out.println(food[i]);
    food[i].a();
    System.out.println();     // to end the line of output
    food[i].b();
    System.out.println();     // to end the line of output
    System.out.println();
}
```

The class diagram

- The following diagram depicts the class hierarchy:



- Notice that `Ham`'s `a` method calls `b`. `Lamb` overrides `b`.
  - This means that calling `a` on a `Lamb` will also have a new result.

```
public class Ham {
    public void a() {
        System.out.print("Ham a   ");
        b();
    }
    public void b() {
        System.out.print("Ham b   ");
    }
    public String toString() {
        return "Ham";
    }
}
public class Lamb extends Ham {
    public void b() {
        System.out.print("Lamb b   ");
    }
}
```

Polymorphism at work!

- `Lamb`'s `a` output: `Ham a  Lamb b`

The table

- Fill out the following table with each class's behavior:

method	Ham	Lamb	Yam	Spam
`a`				
`b`				
`toString`				

The answer

```
Ham[] food = {new Spam(), new Yam(), new Ham(), new Lamb()};
for (int i = 0; i < food.length; i++) {
    System.out.println(food[i]);
    food[i].a();
    food[i].b();
    System.out.println();
}
```

Output:

```
Yam
Yam a    Ham a    Spam b
Spam b

Yam
Yam a    Ham a    Lamb b
Lamb b

Ham
Ham a    Ham b
Ham b

Ham
Ham a    Lamb b
Lamb b
```

Acknowledgments

- The course material used to prepare this presentation is mostly taken/adopted from the list below:
  - Java - How to Program, Paul Deitel and Harvey Deitel, Prentice Hall, 2012
  - Java - An Introduction to Problem Solving and Programming, Walter Savitch, Pearson, 2012
  - Reges/Stepp. *Building Java Programs: A Back to Basics Approach, 3rd edition*
  - Mike Scott, CS314 Course notes, University of Texas Austin

BBM 102 – Introduction to Programming II

Spring 2017

Abstract Classes and Interfaces

Instructors: *Ayça Tarhan, Fuat Akal, Gönenç Ercan, Vahid Garousi*

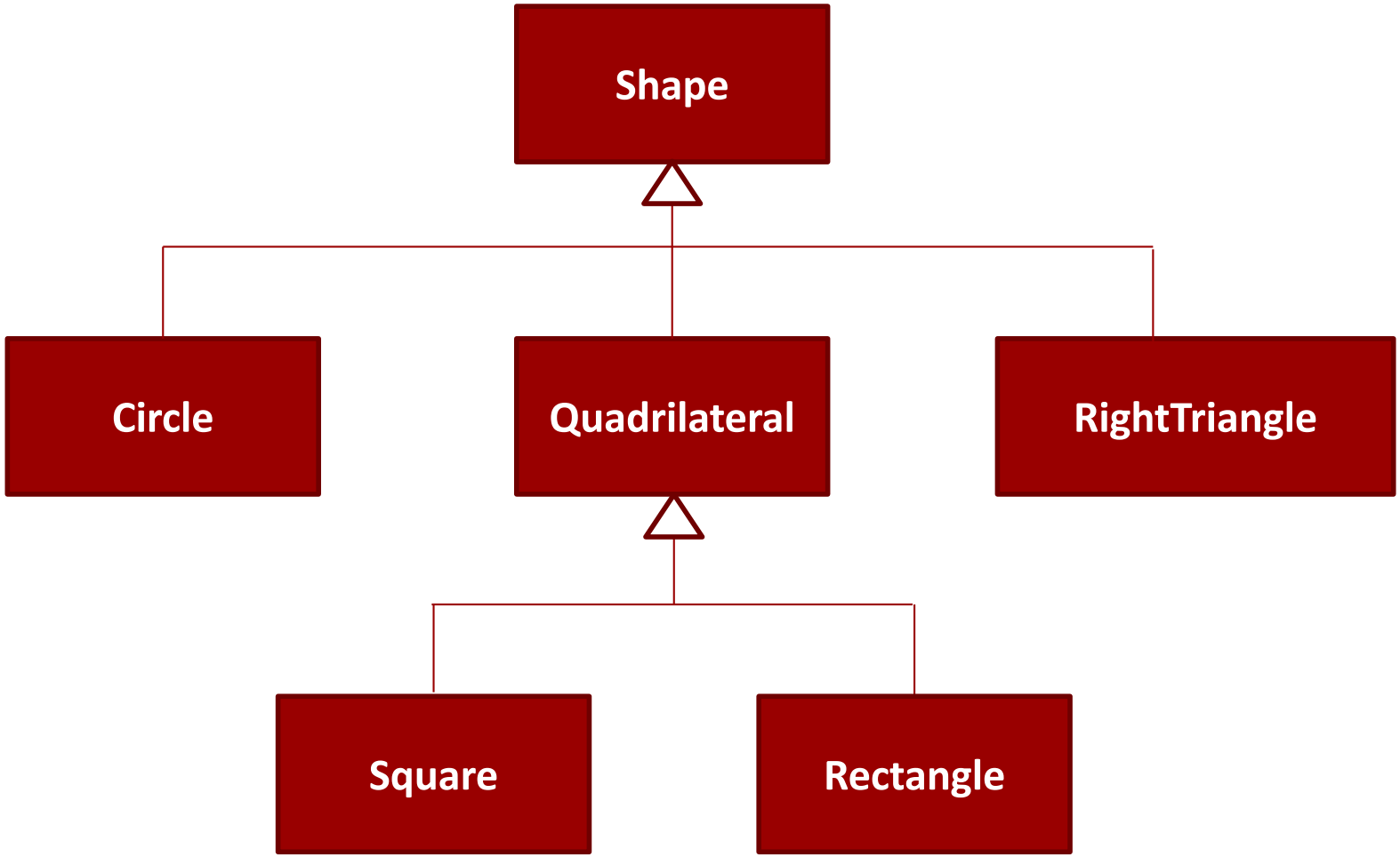
Today

- Abstract Classes
  - Abstract methods
  - Polymorphism with abstract classes
  - Example project: Payroll System
- Interfaces
  - What is an Interface?
  - Defining an Interface
  - Implementing an Interface
  - Implementing Multiple Interfaces
  - Extending a Class and Implementing Interface(s)
  - Extending an Interface
  - Interfaces as Types
- Interfaces vs Abstract Classes

Abstract Classes

- An *abstract class* is a class that is declared **abstract**
- An *abstract class* may or may not include abstract methods.
- Abstract classes cannot be instantiated, but they can be subclassed.

Abstract Classes: Revisiting the Shapes



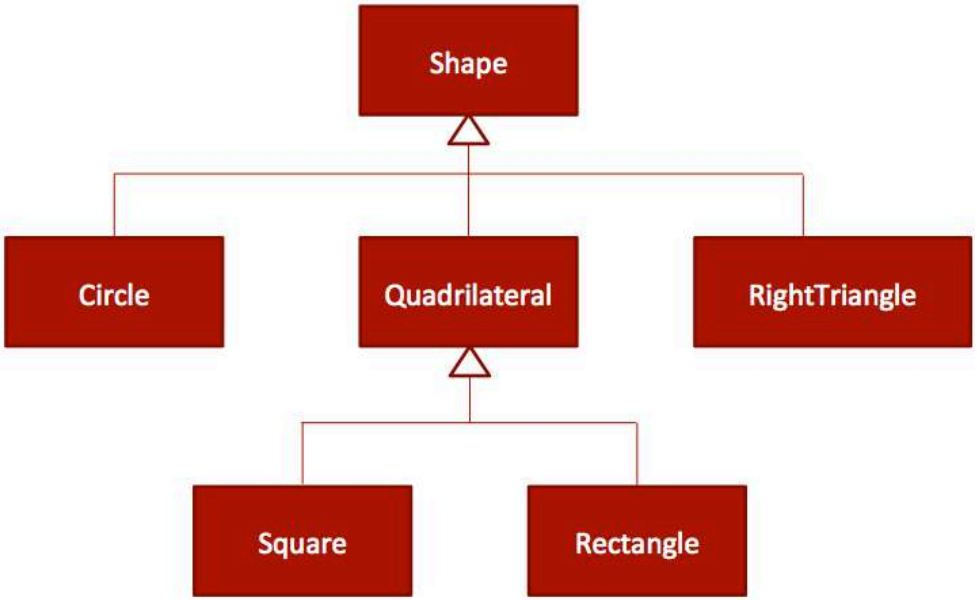
Abstract Classes

- Shapes all have certain states (for example: position, orientation, line color, fill color) and behaviors (for example: moveTo, rotate, resize, draw) in common.

- Some of these states and behaviors are the same for all shapes (for example: position, fill color, and moveTo).

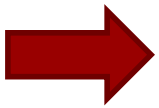
- Others require different implementations (for example, resize or draw).

- All Shapes must be able to draw or resize themselves; they just differ in how they do it.



Abstract Classes

```
public class Shape {
    private String name;

    public Shape(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void draw() {
        // what is the shape?
        // Code...?! Nothing!
    }
}
```



```
public abstract class Shape {
    private String name;

    public Shape(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public abstract void draw();
}
```

Abstract Methods

- An *abstract method* is a method that is declared without an implementation
  - without braces, and followed by a semicolon, like this:

```
public abstract void draw();
```

- When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class.
  - However, if it does not, then the subclass must also be declared abstract.

Abstract Classes

```
public class RightTriangle extends Shape {
    private int a;

    public RightTriangle(String name, int a) {
        super(name);
        this.a = a;
    }

    public int getA() {
        return a;
    }
    // override abstract method
    public void draw() {
        for (int line = 1; line <= a; line++) {
            for (int i = 0; i < line; i++) {
                System.out.print("*");
            }
            System.out.println();
        }
    }
}
```

Abstract Classes

```
public abstract class Quadrilateral
                    extends Shape {

    public Quadrilateral(String name) {
            super(name);
    }

    // still nothing to draw!
    public abstract void draw();
}
```

```
public class Square extends Quadrilateral {
    private int a;

    public Square(String name, int a) {
            super(name);
            this.a = a;
    }
    public int getA() {
            return a;
    }

    // override abstract method
    public void draw() {
            for (int line = 0; line < a; line++) {
                for (int col = 0; col < a; col++) {
                        System.out.print("*");
                }
                System.out.println();
            }
    }
}
```

Abstract Classes

```
public class Program {

    public static void main(String[] args) {
        // compilation error!: "Cannot instantiate the type Shape"
        Shape shape = new Shape("Shape");

        // compilation error!: "Cannot instantiate the type Quadrilateral"
        Quadrilateral quadrilateral = new Quadrilateral("Quadrilateral");

        Square s = new Square("Square", 4);
        s.draw();

        Rectangle r = new Rectangle("Rectangle", 3, 7);
        r.draw();

        RightTriangle t = new RightTriangle("RightTriangle", 5);
        t.draw();
    }
}
```

Abstract Classes

- Are part of the inheritance hierarchy

  Circle extends Shape

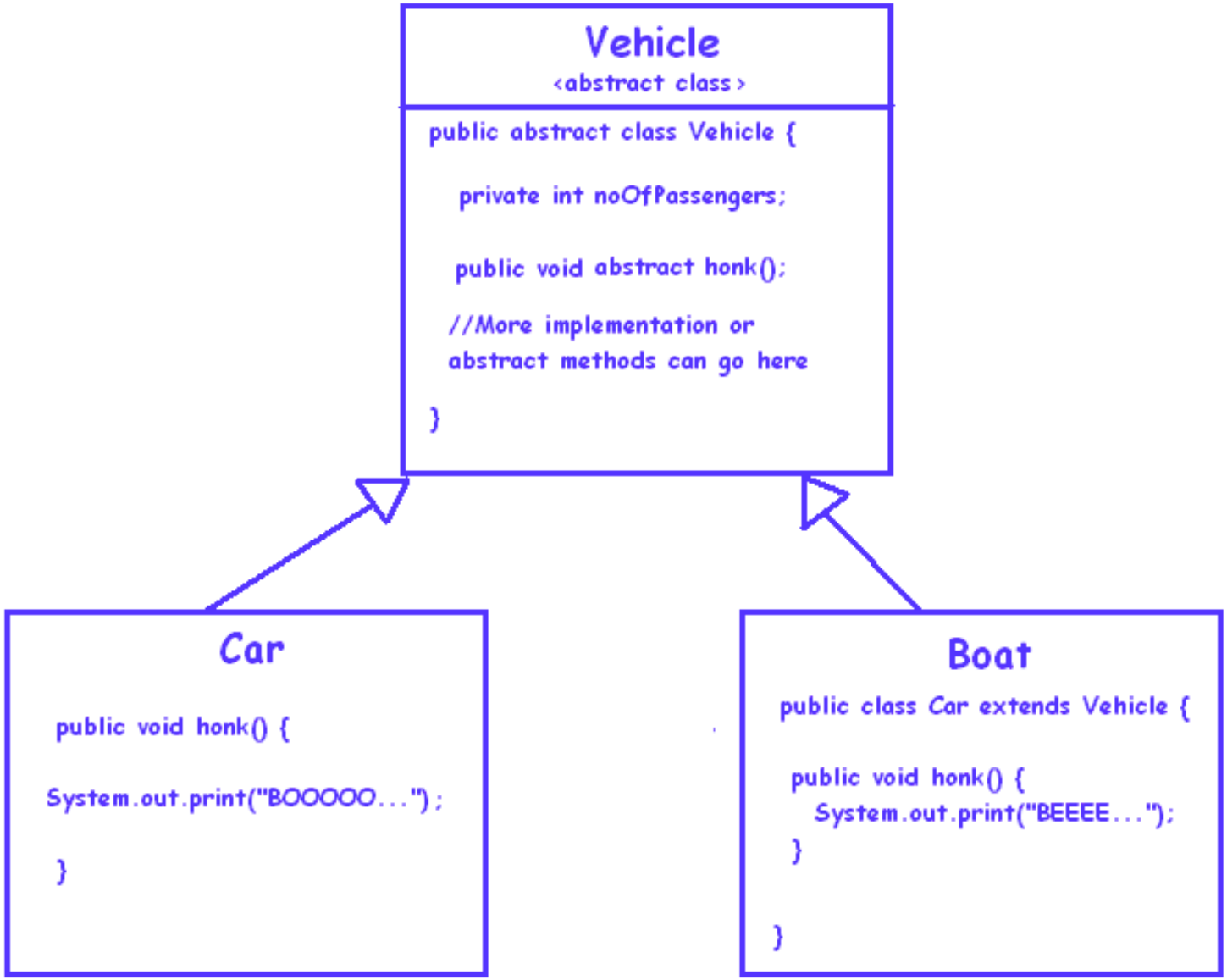
  Square extends Quadrilateral

- Can have constructor(s), but no objects of these classes can be created

```
Shape shape = new Shape("Shape");
// compilation error!: "Cannot instantiate the type Shape“
```

- Classes that can be used to instantiate objects are called **concrete classes**.

Example-1


Vehicle
<abstract class>
public abstract class Vehicle {
private int noOfPassengers;
public void abstract honk();
//More implementation or
abstract methods can go here
}
Car
public void honk() {
System.out.print("BOOOOO...");
}
Boat
public class Car extends Vehicle {
public void honk() {
System.out.print("BEEEE...");
}
}

Example-2

- Imagine there are several instruments, either stringed or wind.
- Design a class hierarchy for only two types of instruments, guitars and flutes.
- You have to design your model in a way that new instruments can be added in the hierarchy later on.
- Imagine there is only one feature for each instrument at the moment, which is the play feature.

Example-2

Abstract class

```
public abstract class Instrument {

    protected String name;
    abstract public void play();
}
```

Still abstract

```
abstract class StringedInstrument extends Instrument {
    protected int numberOfStrings;
}
```

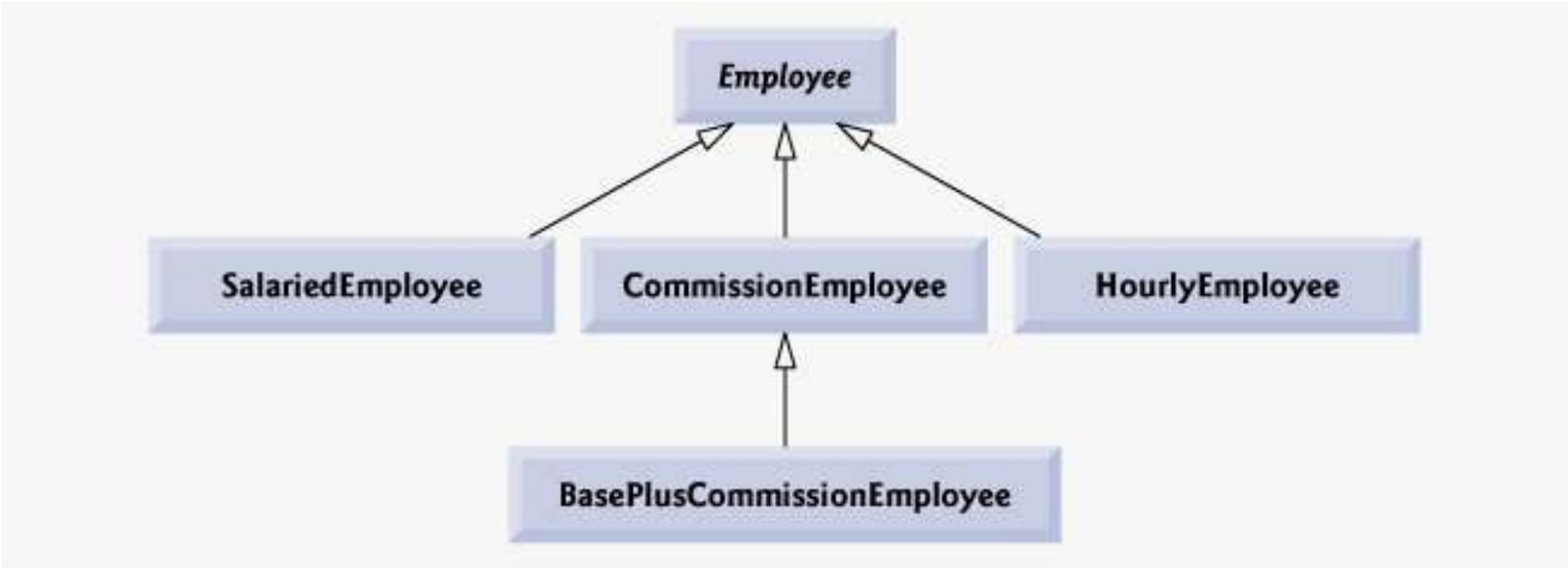
```
public class Guitar extends StringedInstrument{

    public void play(){
       System.out.println("Guitar is rocking!");
    }
}
```

Example-2

```
abstract class WindInstrument extends Instrument {
    //features
}
```

```
public class Flute extends WindInstrument{

    public void play(){
        System.out.println("Flute is rocking!");
    }
}
```

Example Project: Payroll System



Overview of the classes

	earnings	toString
Employee	abstract	*firstName lastName* social security number: *SSN*
Salaried-Employee	weeklySalary	salaried employee: *firstName lastName* social security number: *SSN* weekly salary: *weeklysalary*
Hourly-Employee	*If hours <= 40* wage * hours *If hours > 40* 40 * wage + (hours - 40) * wage * 1.5	hourly employee: *firstName lastName* social security number: *SSN* hourly wage: *wage*; hours worked: *hours*
Commission-Employee	commissionRate * grossSales	commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*
BasePlus-Commission-Employee	(commissionRate * grossSales) + baseSalary	base salaried commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*; base salary: *baseSalary*

Employee.java (1)

```java
// Fig. 10.4: Employee.java
// Employee abstract superclass.

public abstract class Employee
{
   private String firstName;
   private String lastName;
   private String socialSecurityNumber;

   // three-argument constructor
   public Employee( String first, String last, String ssn )
   {
      firstName = first;
      lastName = last;
      socialSecurityNumber = ssn;
   } // end three-argument Employee constructor

   // set first name
   public void setFirstName( String first )
   {
      firstName = first;
   } // end method setFirstName

   // return first name
   public String getFirstName()
   {
      return firstName;
   } // end method getFirstName

   // set last name
   public void setLastName( String last )
   {
      lastName = last;
   } // end method setLastName

```

Employee.java (2)

```
36      // return last name
37      public String getLastName()
38      {
39         return lastName;
40      } // end method getLastName
41
42      // set social security number
43      public void setSocialSecurityNumber( String ssn )
44      {
45         socialSecurityNumber = ssn; // should validate
46      } // end method setSocialSecurityNumber
47
48      // return social security number
49      public String getSocialSecurityNumber()
50      {
51         return socialSecurityNumber;
52      } // end method getSocialSecurityNumber
53
54      // return String representation of Employee object
55      public String toString()
56      {
57         return String.format( "%s %s\nsocial security number: %s",
58            getFirstName(), getLastName(), getSocialSecurityNumber()
59      } // end method toString
60
61      // abstract method overridden by subclasses
62      public abstract double earnings(); // no implementation here
63   } // end abstract class Employee
```

Earnings will be calculated in subclasses

SalariedEmployee.java

```java
public class SalariedEmployee extends Employee
{
   private double weeklySalary;

   // four-argument constructor
   public SalariedEmployee( String first, String last, String ssn,
      double salary )
   {
      super( first, last, ssn ); // pass to Employee constructor
      setWeeklySalary( salary ); // validate and store salary
   } // end four-argument SalariedEmployee constructor

   // set salary
   public void setWeeklySalary( double salary )
   {
      weeklySalary = salary < 0.0 ? 0.0 : salary;
   } // end method setWeeklySalary

   // return salary
   public double getWeeklySalary()
   {
      return weeklySalary;
   } // end method getWeeklySalary

   // calculate earnings; override abstract method earnings in Employee
   public double earnings()
   {
      return getWeeklySalary();
   } // end method earnings

   // return String representation of SalariedEmployee object
   public String toString()
   {
      return String.format( "salaried employee: %s\n%s: $%,.2f",
         super.toString(), "weekly salary", getWeeklySalary() );
   } // end method toString
} // end class SalariedEmployee
```

Overridden methods

HourlyEmployee.java (1)

```
4   public class HourlyEmployee extends Employee
5   {
6      private double wage; // wage per hour
7      private double hours; // hours worked for week
8
9      // five-argument constructor
10     public HourlyEmployee( String first, String last, String ssn,
11        double hourlyWage, double hoursWorked )
12     {
13        super( first, last, ssn );
14        setWage( hourlyWage ); // validate hourly wage
15        setHours( hoursWorked ); // validate hours worked
16     } // end five-argument HourlyEmployee constructor
17
18     // set wage
19     public void setWage( double hourlyWage )
20     {
21        wage = ( hourlyWage < 0.0 ) ? 0.0 : hourlyWage;
22     } // end method setWage
23
24     // return wage
25     public double getWage()
26     {
27        return wage;
28     } // end method getWage
29
30     // set hours worked
31     public void setHours( double hoursWorked )
32     {
33        hours = ( ( hoursWorked >= 0.0 ) && ( hoursWorked <= 168.0 ) ) ?
34           hoursWorked : 0.0;
35     } // end method setHours
```

HourlyEmployee.java (2)

```java

    // return hours worked
    public double getHours()
    {
       return hours;
    } // end method getHours

    // calculate earnings; override abstract method earnings in Employee
    public double earnings()
    {
       if ( getHours() <= 40 ) // no overtime
          return getWage() * getHours();
       else
          return 40 * getWage() + ( gethours() - 40 ) * getWage() * 1.5;
    } // end method earnings

    // return String representation of HourlyEmployee object
    public String toString()
    {
       return String.format( "hourly employee: %s\n%s: $%,.2f; %s: %,.2f",
          super.toString(), "hourly wage", getWage(),
          "hours worked", getHours() );
    } // end method toString
 } // end class HourlyEmployee
```

CommissionEmployee.java (1)

```
4   public class CommissionEmployee extends Employee
5   {
6      private double grossSales; // gross weekly sales
7      private double commissionRate; // commission percentage
8
9      // five-argument constructor
10     public CommissionEmployee( String first, String last, String ssn,
11        double sales, double rate )
12     {
13        super( first, last, ssn );
14        setGrossSales( sales );
15        setCommissionRate( rate );
16     } // end five-argument CommissionEmployee constructor
17
18     // set commission rate
19     public void setCommissionRate( double rate )
20     {
21        commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
22     } // end method setCommissionRate
23
24     // return commission rate
25     public double getCommissionRate()
26     {
27        return commissionRate;
28     } // end method getCommissionRate
29
30     // set gross sales amount
31     public void setGrossSales( double sales )
32     {
33        grossSales = ( sales < 0.0 ) ? 0.0 : sales;
34     } // end method setGrossSales
```

CommissionEmployee.java (2)

```
    // return gross sales amount
    public double getGrossSales()
    {
       return grossSales;
    } // end method getGrossSales

    // calculate earnings; override abstract method earnings in Employee
    public double earnings()
    {
       return getCommissionRate() * getGrossSales();
    } // end method earnings

    // return String representation of CommissionEmployee object
    public String toString()
    {
       return String.format( "%s: %s\n%s: $%,.2f; %s: %.2f",
          "commission employee", super.toString(),
          "gross sales", getGrossSales(),
          "commission rate", getCommissionRate() );
    } // end method toString
 } // end class CommissionEmployee
```

BasePlusCommissionEmployee.java

```java
public class BasePlusCommissionEmployee extends CommissionEmployee
{
   private double baseSalary; // base salary per week

   // six-argument constructor
   public BasePlusCommissionEmployee( String first, String last,
      String ssn, double sales, double rate, double salary )
   {
      super( first, last, ssn, sales, rate );
      setBaseSalary( salary ); // validate and store base salary
   } // end six-argument BasePlusCommissionEmployee constructor

   // set base salary
   public void setBaseSalary( double salary )
   {
      baseSalary = ( salary < 0.0 ) ? 0.0 : salary; // non-negative
   } // end method setBaseSalary

   // return base salary
   public double getBaseSalary()
   {
      return baseSalary;
   } // end method getBaseSalary

   // calculate earnings; override method earnings in CommissionEmployee
   public double earnings()
   {
      return getBaseSalary() + super.earnings();
   } // end method earnings

   // return String representation of BasePlusCommissionEmployee object
   public String toString()
   {
      return String.format( "%s %s; %s: $%,.2f",
         "base-salaried", super.toString(),
         "base salary", getBaseSalary() );
   } // end method toString
} // end class BasePlusCommissionEmployee
```

PayrollSystemTest.java (1)

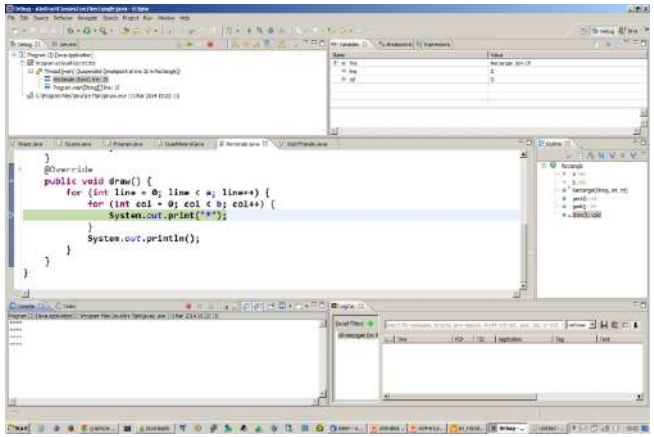
```
4   public class PayrollSystemTest
5   {
6      public static void main( String args[] )
7      {
8         // create subclass objects
9         SalariedEmployee salariedEmployee =
10           new SalariedEmployee( "John", "Smith", "111-11-1111", 800.00 );
11        HourlyEmployee hourlyEmployee =
12           new HourlyEmployee( "Karen", "Price", "222-22-2222", 16.75, 40 );
13        CommissionEmployee commissionEmployee =
14           new CommissionEmployee(
15           "Sue", "Jones", "333-33-3333", 10000, .06 );
16        BasePlusCommissionEmployee basePlusCommissionEmployee =
17           new BasePlusCommissionEmployee(
18           "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );
19
20        System.out.println( "Employees processed individually:\n" );
21
22        System.out.printf( "%s\n%s: $%,.2f\n\n",
23           salariedEmployee, "earned", salariedEmployee.earnings() );
24        System.out.printf( "%s\n%s: $%,.2f\n\n",
25           hourlyEmployee, "earned", hourlyEmployee.earnings() );
26        System.out.printf( "%s\n%s: $%,.2f\n\n",
27           commissionEmployee, "earned", commissionEmployee.earnings() );
28        System.out.printf( "%s\n%s: $%,.2f\n\n",
29           basePlusCommissionEmployee,
30           "earned", basePlusCommissionEmployee.earnings() );
31
32        // create four-element Employee array
33        Employee employees[] = new Employee[ 4 ];
34
35        // initialize array with Employees
36        employees[ 0 ] = salariedEmployee;
37        employees[ 1 ] = hourlyEmployee;
38        employees[ 2 ] = commissionEmployee;
39        employees[ 3 ] = basePlusCommissionEmployee;
```

PayrollSystemTest.java (2)

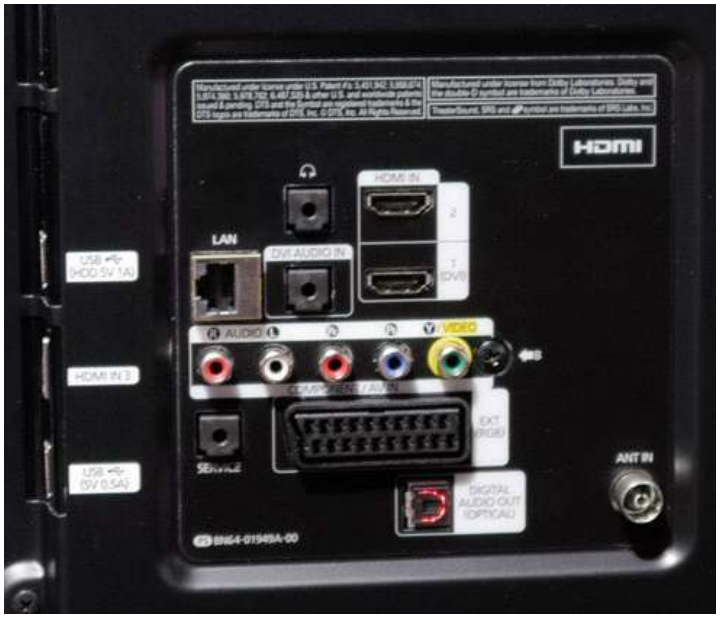
```
41         System.out.println( "Employees processed polymorphically:\n" );
42
43         // generically process each element in array employees
44         for ( Employee currentEmployee : employees )
45         {
46            System.out.println( currentEmployee ); // invokes toString
47
48            // determine whether element is a BasePlusCommissionEmployee
49            if ( currentEmployee instanceof BasePlusCommissionEmployee )
50            {
51               // downcast Employee reference to
52               // BasePlusCommissionEmployee reference
53               BasePlusCommissionEmployee employee =
54                  ( BasePlusCommissionEmployee ) currentEmployee;
55
56               double oldBaseSalary = employee.getBaseSalary();
57               employee.setBaseSalary( 1.10 * oldBaseSalary );
58               System.out.printf(
59                  "new base salary with 10%% increase is: $%,.2f\n",
60                  employee.getBaseSalary() );
61            } // end if
62
63            System.out.printf(
64               "earned $%,.2f\n\n", currentEmployee.earnings() );
65         } // end for
66
67         // get type name of each object in employees array
68         for ( int j = 0; j < employees.length; j++ )
69            System.out.printf( "Employee %d is a %s\n", j,
70               employees[ j ].getClass().getName() );
71      } // end main
72   } // end class PayrollSystemTest
```

Interfaces

GUI



LCD/LED TV



Laptop

Concept of Interface

- An interface is a contract. It guarantees that the system will have certain functionalities.
- An interface is an integration point between two systems.
- A system can have many interfaces, so it can be integrated to many other systems.

Defining an Interface

- Keyword `interface` is used to define an interface
- Methods in an interface must be `public` and `abstract`, these keywords are commonly omitted
- Interfaces can include `public static final` variables (constants), these keywords are commonly omitted

```
public interface Shape {
    public abstract void draw();

    public static final double PI = 3.14;
}
```

No need to write

Implementing an Interface

- An interface is implemented by the keyword `implements`
- Any class implementing an interface must either implement all methods of it, or be declared abstract

```
public class RightTriangle implements Shape {
    // .....
    public void draw() {
        for (int line = 1; line <= a; line++) {
            for (int i = 0; i < line; i++) {
                System.out.print("*");
            }
            System.out.println();
        }
    }
}
```

Implementing Multiple Interfaces

- More than one interface can be implemented by a class.
- Names of interfaces are separated by comma

```
public class LedTv implements Usb, Hdmi, Scart, Vga {

    // .....

}
```

Question: What if at least two interfaces include the same method definition?

Extending a Class and Implementing Interface(s)

```
public class Car extends Vehicle
                        implements Shape {

        public void draw() {
            // ....
        }
}
```

Extending an Interface

- It is possible for an interface to extend another interface

```
public interface I1 {
    void m1();
}
```

```
public interface I2 extends I1 {
    void m2();
}
```

```
public class C1 implements I1 {

    public void m1() {
        // ...
    }
}
```

```
public class C2 implements I2 {
    public void m1() {
        // ...
    }
    public void m2() {
        // ...
    }
}
```

Interfaces as Types

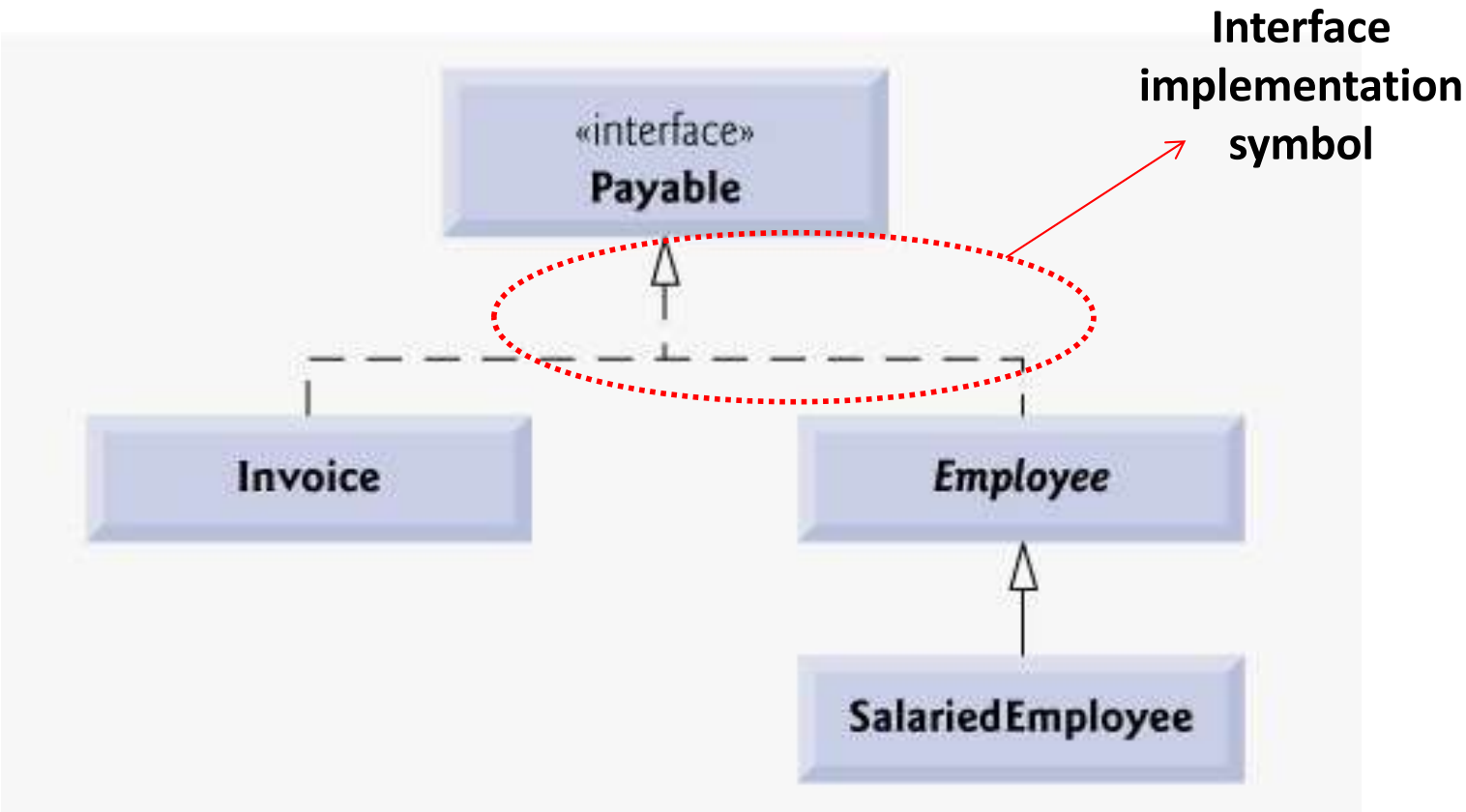
- When you define a new interface, you are defining a new reference data type.
- You can use interface names anywhere you can use any other data type name.
- If you define a reference variable whose type is an interface, any object you assign to it must be an instance of a class that implements the interface.

Interfaces as Types

```
public class Program {
    public static void main(String[] args) {
        Shape shape;

        shape = new Square(4);
        shape.draw();

        shape = new Rectangle(3, 7);
        shape.draw();

        shape = new RightTriangle(5);
        shape.draw();
    }
}
```

```
public class Program {
    public static void main(String[] args) {
        Shape[] shapes = new Shape[3];
        shapes[0] = new Square(5);
        shapes[1] = new Rectangle(2, 8);
        shapes[2] = new RightTriangle(3);
        for (Shape s : shapes) {
            drawIt(s);
        }
    }

    public static void drawIt(Shape s) {
        s.draw();
    }
}
```

Example Project: Payroll System Revisited



Payable.java

```java
// Fig. 10.11: Payable.java
// Payable interface declaration.

public interface Payable
{
   double getPaymentAmount(); // calculate payment; no implementation
} // end interface Payable
```

Invoice.java (1)

```
4   public class Invoice implements Payable
5   {
6      private String partNumber;
7      private String partDescription;
8      private int quantity;
9      private double pricePerItem;
10
11     // four-argument constructor
12     public Invoice( String part, String description, int count,
13        double price )
14     {
15        partNumber = part;
16        partDescription = description;
17        setQuantity( count ); // validate and store quantity
18        setPricePerItem( price ); // validate and store price per item
19     } // end four-argument Invoice constructor
20
21     // set part number
22     public void setPartNumber( String part )
23     {
24        partNumber = part;
25     } // end method setPartNumber
26
27     // get part number
28     public String getPartNumber()
29     {
30        return partNumber;
31     } // end method getPartNumber
32
33     // set description
34     public void setPartDescription( String description )
35     {
36        partDescription = description;
37     } // end method setPartDescription
38
39     // get description
40     public String getPartDescription()
41     {
42        return partDescription;
43     } // end method getPartDescription
```

Invoice.java (2)

```
45      // set quantity
46      public void setQuantity( int count )
47      {
48         quantity = ( count < 0 ) ? 0 : count; // quantity cannot be negative
49      } // end method setQuantity
50
51      // get quantity
52      public int getQuantity()
53      {
54         return quantity;
55      } // end method getQuantity
56
57      // set price per item
58      public void setPricePerItem( double price )
59      {
60         pricePerItem = ( price < 0.0 ) ? 0.0 : price; // validate price
61      } // end method setPricePerItem
62
63      // get price per item
64      public double getPricePerItem()
65      {
66         return pricePerItem;
67      } // end method getPricePerItem
68
69      // return String representation of Invoice object
70      public String toString()
71      {
72         return String.format( "%s: \n%s: %s (%s) \n%s: %d \n%s: $%,.2f",
73            "invoice", "part number", getPartNumber(), getPartDescription(),
74            "quantity", getQuantity(), "price per item", getPricePerItem() );
75      } // end method toString
76
77      // method required to carry out contract with interface Payable
78      public double getPaymentAmount()
79      {
80         return getQuantity() * getPricePerItem(); // calculate total cost
81      } // end method getPaymentAmount
82   } // end class Invoice
```

Employee.java

`Payable` **interface includes** `getPaymentAmount()` **method, but** `Employee` **class does not implement it!**

```
4   public abstract class Employee implements Payable
5   {
6      private String firstName;
7      private String lastName;
8      private String socialSecurityNumber;
9
10     // three-argument constructor
11     public Employee( String first, String last, String ssn )
12     {
13        firstName = first;
14        lastName = last;
15        socialSecurityNumber = ssn;
16     } // end three-argument Employee constructor
17
18        /* Rest of the class is same as the previous example
19        except there is no earnings() method! */
20
21
22
23
```

SalariedEmployee.java

```
 4  public class SalariedEmployee extends Employee
 5  {
 6     private double weeklySalary;
 7
 8     // four-argument constructor
 9     public SalariedEmployee( String first, String last, String ssn,
10        double salary )
11     {
12        super( first, last, ssn ); // pass to Employee constructor
13        setWeeklySalary( salary ); // validate and store salary
14     } // end four-argument SalariedEmployee constructor
15
16     // set salary
17     public void setWeeklySalary( double salary )
18     {
19        weeklySalary = salary < 0.0 ? 0.0 : salary;
20     } // end method setWeeklySalary
21
22     // return salary
23     public double getWeeklySalary()
24     {
25        return weeklySalary;
26     } // end method getWeeklySalary
27
28     // calculate earnings; implement interface Payable method that was
29     // abstract in superclass Employee
30     public double getPaymentAmount()
31     {
32        return getWeeklySalary();
33     } // end method getPaymentAmount
34
35     // return String representation of SalariedEmployee object
36     public String toString()
37     {
38        return String.format( "salaried employee: %s\n%s: $%,.2f",
39           super.toString(), "weekly salary", getWeeklySalary() );
40     } // end method toString
41  } // end class SalariedEmployee
```

PayableInterfaceTest.java

```java
public class PayableInterfaceTest
{
   public static void main( String args[] )
   {
      // create four-element Payable array
      Payable payableObjects[] = new Payable[ 4 ];

      // populate array with objects that implement Payable
      payableObjects[ 0 ] = new Invoice( "01234", "seat", 2, 375.00 );
      payableObjects[ 1 ] = new Invoice( "56789", "tire", 4, 79.95 );
      payableObjects[ 2 ] =
         new SalariedEmployee( "John", "Smith", "111-11-1111", 800.00 );
      payableObjects[ 3 ] =
         new SalariedEmployee( "Lisa", "Barnes", "888-88-8888", 1200.00 );

      System.out.println(
         "Invoices and Employees processed polymorphically:\n" );

      // generically process each element in array payableObjects
      for ( Payable currentPayable : payableObjects )
      {
         // output currentPayable and its appropriate payment amount
         System.out.printf( "%s \n%s: $%,.2f\n\n",
            currentPayable.toString(),
            "payment due", currentPayable.getPaymentAmount() );
      } // end for
   } // end main
} // end class PayableInterfaceTest
```

Interfaces vs Abstract Classes

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can have static methods, main method and constructor**.	Interface **can't have static methods, main method or constructor**.
5) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
6) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.

7) **Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Summary

- Abstract class is defined with the keyword `abstract`
- If a class includes an abstract method, it must be declared as abstract
- Objects of abstract classes cannot be created
- Interface is defined with the keyword `interface`
- A class can *implement* an interface, an interface can *extend* an interface
- A class can implement many interfaces
- Objects of interfaces cannot be created

Acknowledgements

- The course material used to prepare this presentation is mostly taken/adopted from the list below:
  - Java - How to Program, Paul Deitel and Harvey Deitel, Prentice Hall, 2012
- http://www.javatpoint.com/difference-between-abstract-class-and-interface

BBM 102 – Introduction to Programming II

Spring 2017

Collections Framework

Instructors: *Ayça Tarhan, Fuat Akal, Gönenç Ercan, Vahid Garousi*

Today

- The **`java.util.Arrays`** class
- Java Collection Framework
  - java.util.Collection interface
  - java.util.List interface
    - java.util.ArrayList class
  - java.util.Set interface
    - java.util.HashSet class
  - java.util.Map interface
    - java.util.HashMap class
    - java.util.Hashtable class
    - java.util.Properties class

java.util.Arrays

- Provides high-level static methods for manipulating arrays, such as:
  - **sort** for sorting an array,
  - **binarySearch** for searching a sorted array,
  - **equals** for comparing arrays
  - **fill** for placing values into an array

Arrays – Example

```
3   import java.util.Arrays;
4
5   public class UsingArrays
6   {
7      private int intArray[] = { 1, 2, 3, 4, 5, 6 };
8      private double doubleArray[] = { 8.4, 9.3, 0.2, 7.9, 3.4 };
9      private int filledIntArray[], intArrayCopy[];
10
11     // constructor initializes arrays
12     public UsingArrays()
13     {
14        filledIntArray = new int [ 10 ]; // create int array with 10 elements
15        intArrayCopy = new int [ intArray.length ];
16
17        Arrays.fill( filledIntArray, 7 ); // fill with 7s
18        Arrays.sort( doubleArray ); // sort doubleArray ascending
19
20        // copy array intArray into array intArrayCopy
21        System.arraycopy( intArray, 0, intArrayCopy,
22           0, intArray.length );
23     } // end UsingArrays constructor
```

Arrays – Example (continued)

```
47      // find value in array intArray
48      public int searchForInt( int value )
49      {
50         return Arrays.binarySearch( intArray, value );
51      } // end method searchForInt
52
53      // compare array contents
54      public void printEquality()
55      {
56         boolean b = Arrays.equals( intArray, intArrayCopy );
57         System.out.printf( "intArray %s intArrayCopy\n",
58            ( b ? "==" : "!=" ) );
59
60         b = Arrays.equals( intArray, filledIntArray );
61         System.out.printf( "intArray %s filledIntArray\n",
62            ( b ? "==" : "!=" ) );
63      } // end method printEquality
```

Arrays – Example (continued)

```
26      public void printArrays()
27      {
28         System.out.print( "doubleArray: " );
29         for ( double doubleValue : doubleArray )
30            System.out.printf( "%.1f ", doubleValue );
31
32         System.out.print( "\nintArray: " );
33         for ( int intValue : intArray )
34            System.out.printf( "%d ", intValue );
35
36         System.out.print( "\nfilledIntArray: " );
37         for ( int intValue : filledIntArray )
38            System.out.printf( "%d ", intValue );
39
40         System.out.print( "\nintArrayCopy: " );
41         for ( int intValue : intArrayCopy )
42            System.out.printf( "%d ", intValue );
43
44         System.out.println( "\n" );
45      } // end method printArrays
```

Arrays – Example (continued)

```
65      public static void main( String args[] )
66      {
67         UsingArrays usingArrays = new UsingArrays();
68
69         usingArrays.printArrays();
70         usingArrays.printEquality();
71
72         int location = usingArrays.searchForInt( 5 );
73         if ( location >= 0 )
74            System.out.printf(
75               "Found 5 at element %d in intArray\n", location );
76         else
77            System.out.println( "5 not found in intArray" );
78
79         location = usingArrays.searchForInt( 8763 );
80         if ( location >= 0 )
81            System.out.printf(
82               "Found 8763 at element %d in intArray\n", location );
83         else
84            System.out.println( "8763 not found in intArray" );
85      } // end main
86   } // end class UsingArrays
```

Java Collections Framework

- Group of objects can be kept in an array, but arrays are not feasible when number of objects increase or decrease during the execution of the program
- The ***Java Collections Framework*** is a collection of interfaces and classes that may be used to manipulate groups of objects
- The classes implemented in the ***Java Collections Framework*** serve as reusable data structures and include algorithms for common tasks such as sorting or searching
- The framework uses parameterized classes so you can use them with the classes of your choice
- The framework is largely contained in the package **`java.util`**

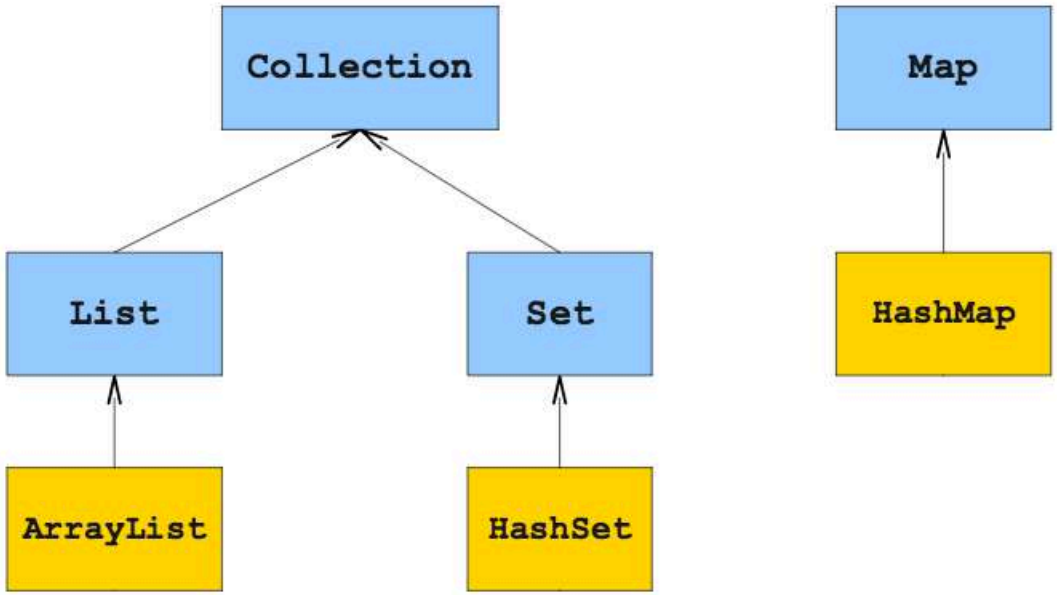
Basic collections

The three major collections in Java:

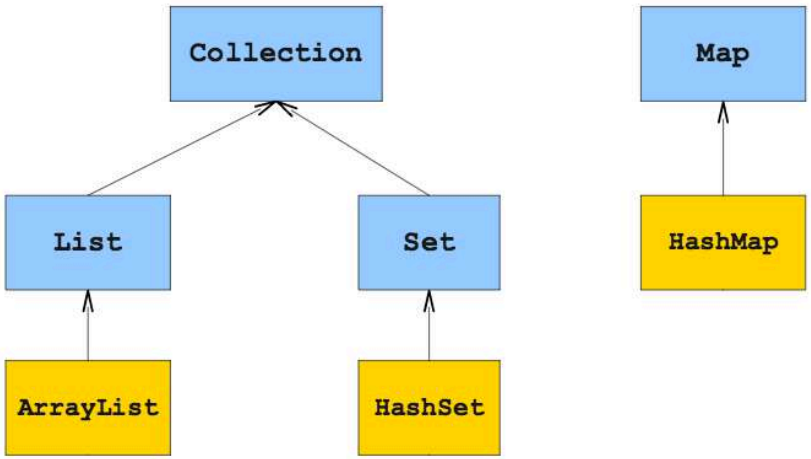
- ***Set***: A collection of elements that is guaranteed to contain no duplicates.
- ***List***: An ordered collection of elements, often accessed by integer indexes or by iteration.
- ***Map***: A collection of key/value pairs in which each key is associated with a corresponding value.

java.util.Collection interface

- It is the highest level of Java's framework for collection classes
- It describes the basic operations that all collection classes (except **`Map`**) should implement (listed in the next page)
- The JDK does not provide any *direct* implementations of this interface: it provides implementations of more specific sub-interfaces like Set and List as in the example diagram below:

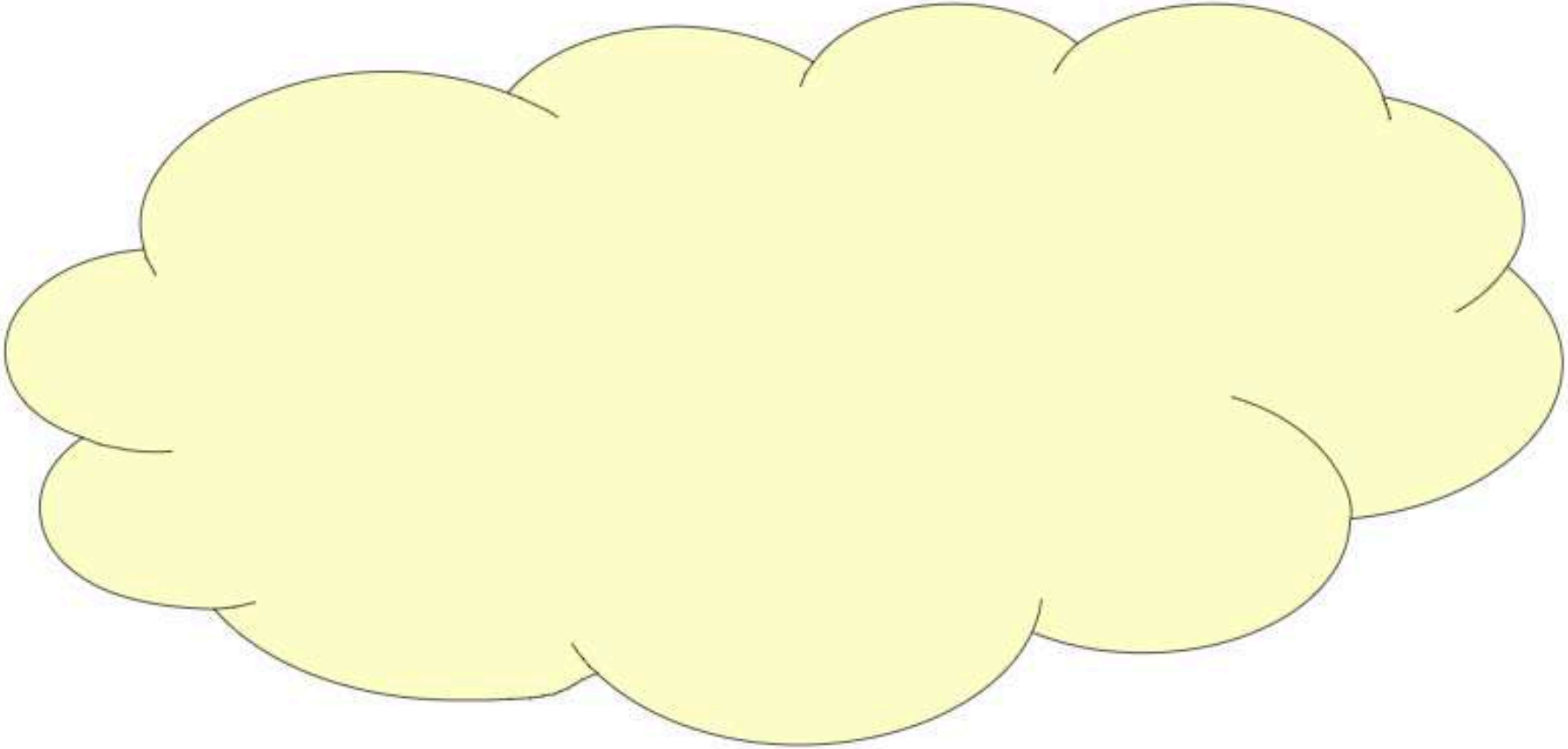


What is a Set?

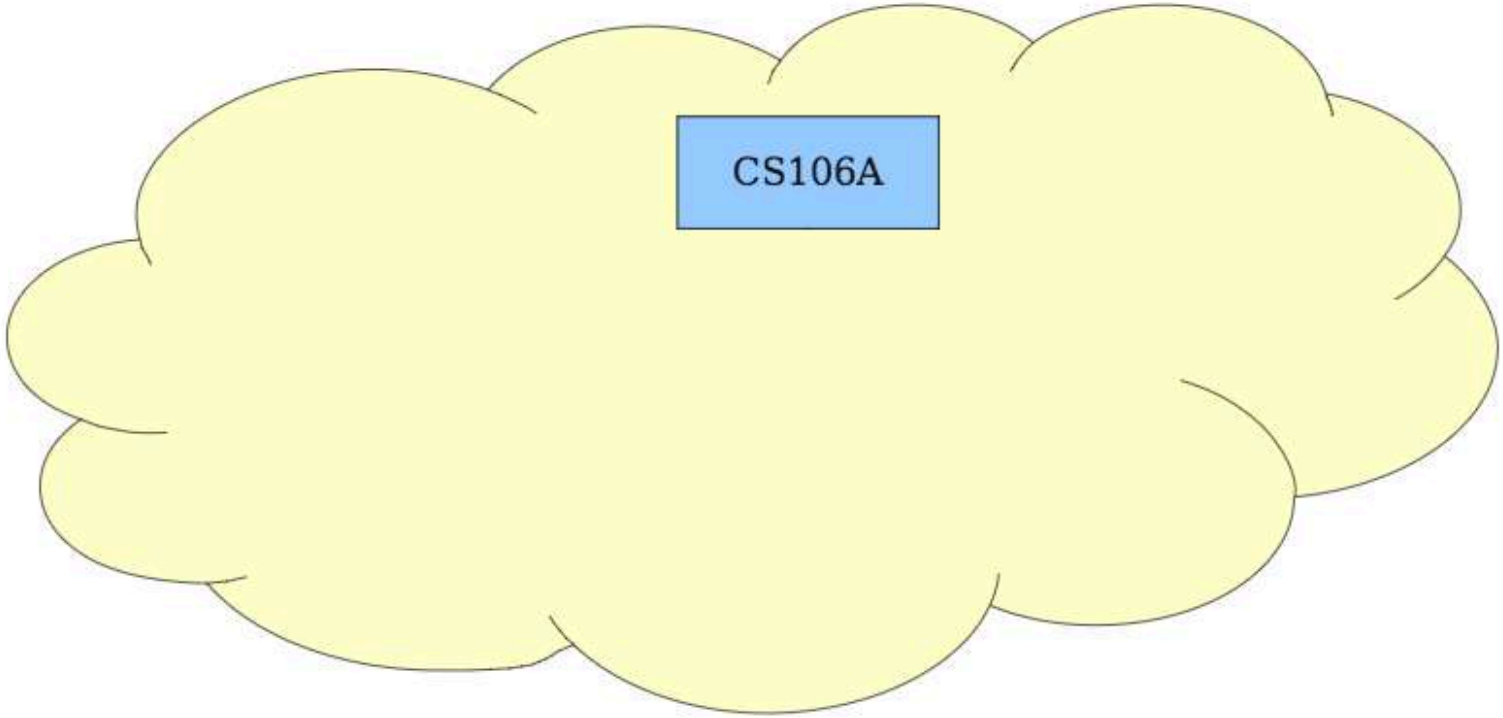


java.util.Set interface

- A **Set** is a **Collection** that contains unique elements (i.e., no duplicate elements).
- The collections framework contains several **Set** implementations, including **HashSet**
- Major operations are:
  - Adding an element
  - Removing an element
  - Checking whether an element exists
- Useful for answering questions of the form “have I seen this before?”



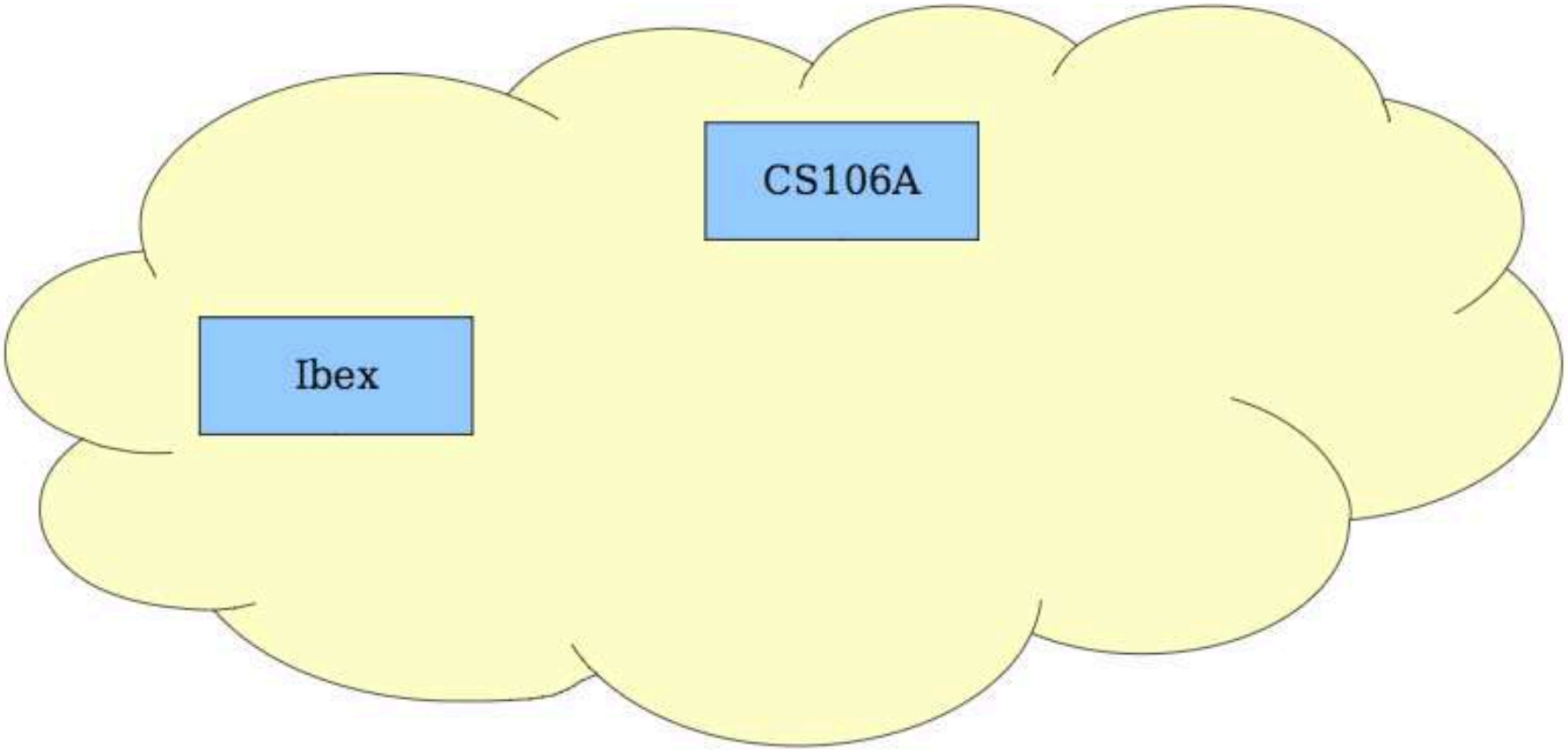
```
HashSet<String> mySet = new HashSet<String>();
```



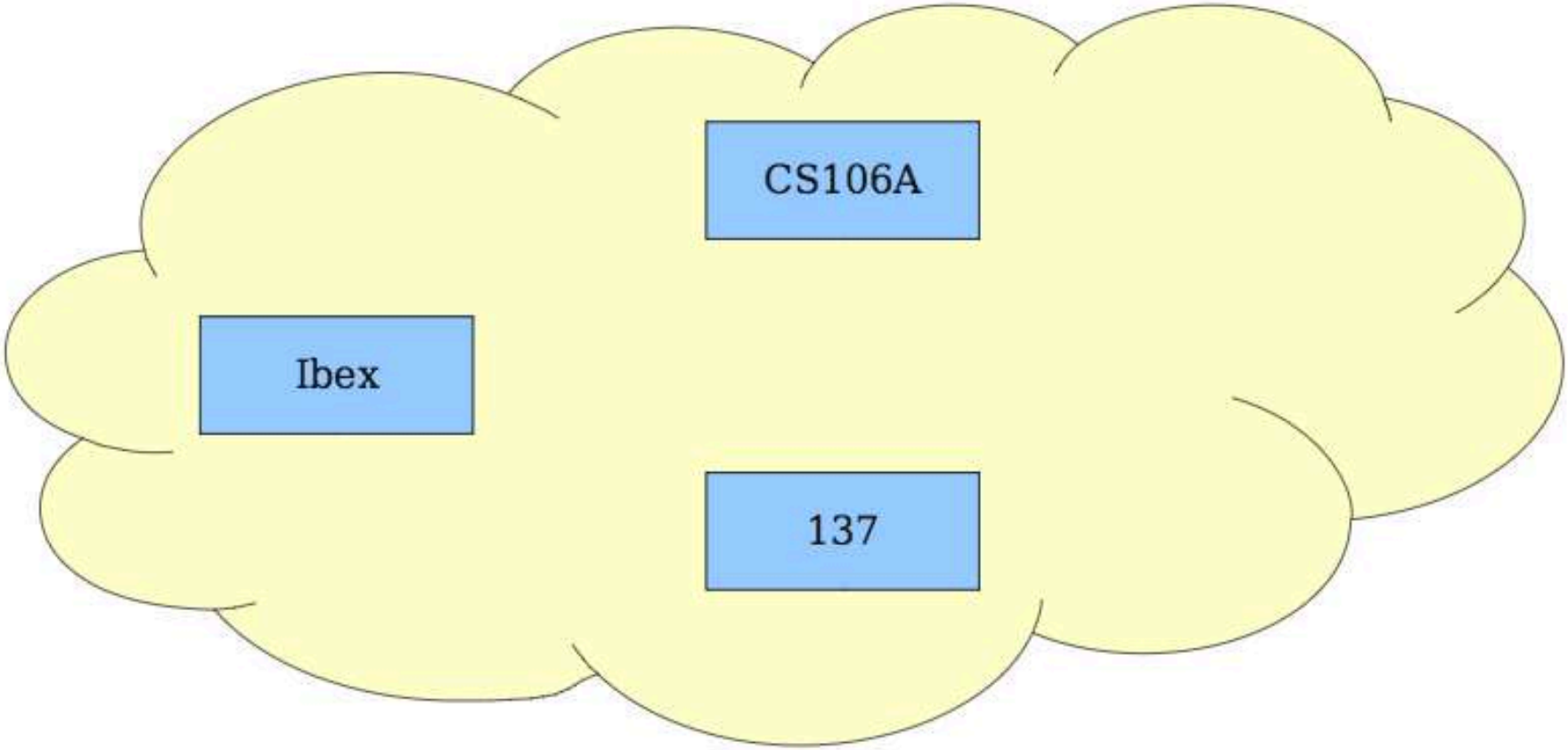

```
HashSet<String> mySet = new HashSet<String>();
mySet.add("CS106A");
```

To add a value to a **HashSet**, use the syntax

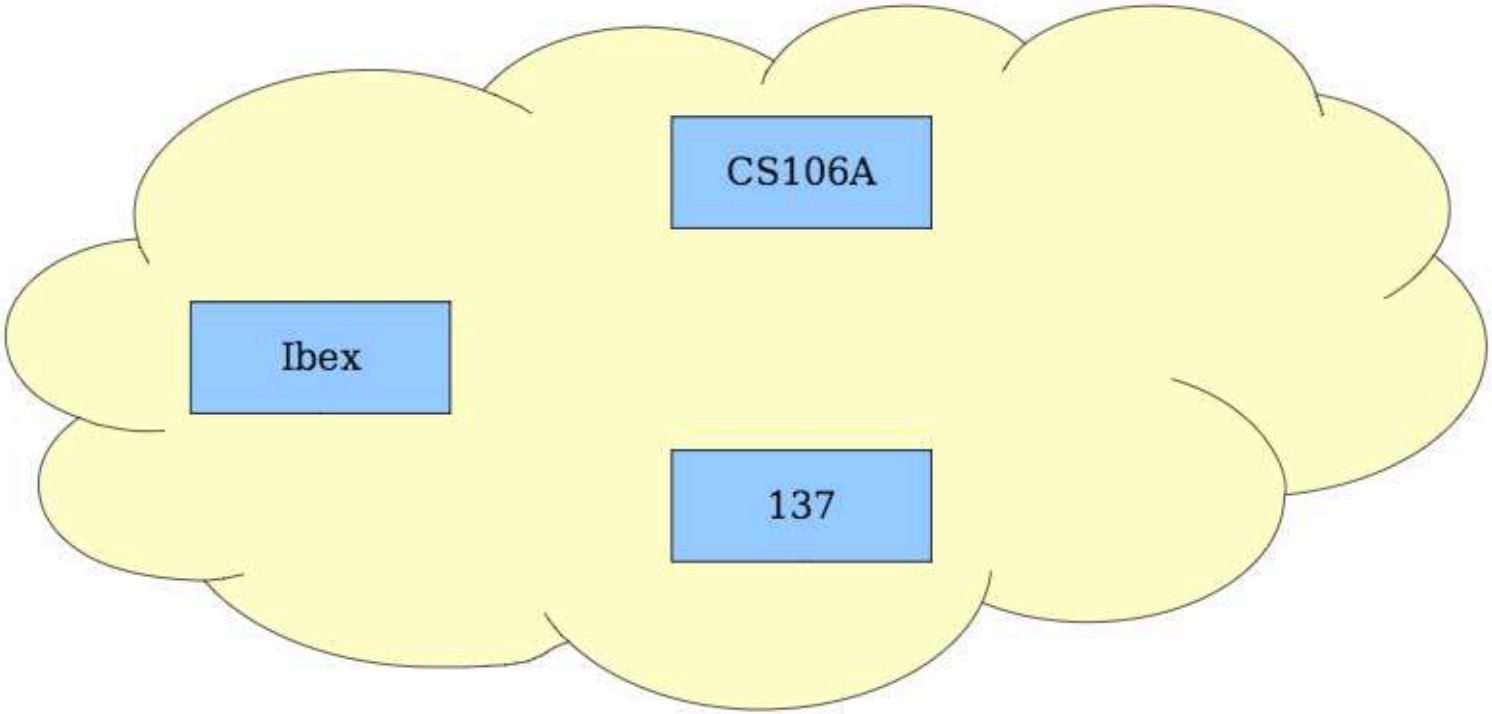
***set*.add(*value*)**




```
HashSet<String> mySet = new HashSet<String>();
mySet.add("CS106A");
mySet.add("Ibex");
```

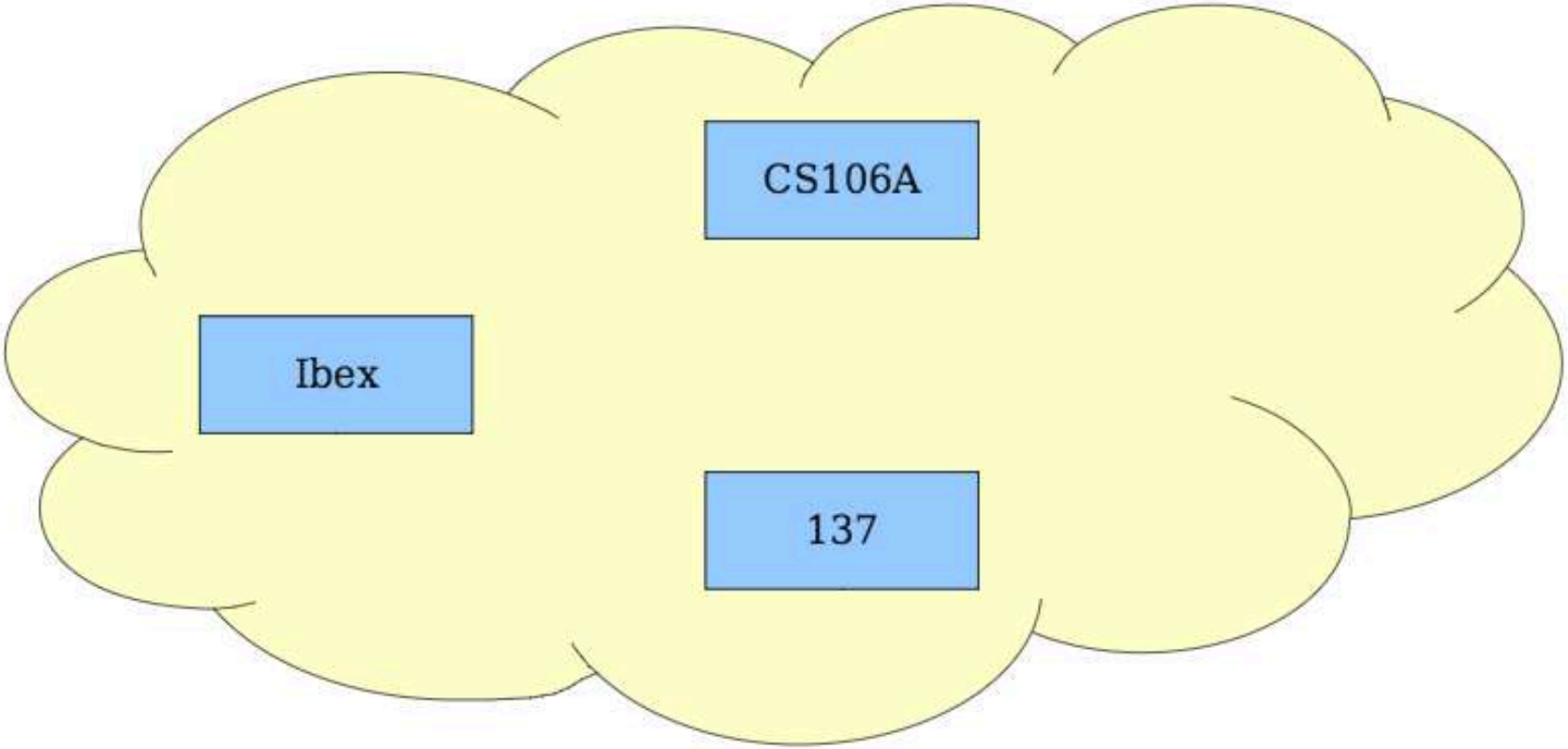



```
HashSet<String> mySet = new HashSet<String>();
mySet.add("CS106A");
mySet.add("Ibex");
mySet.add("137");
```




```
HashSet<String> mySet = new HashSet<String>();
mySet.add("CS106A");
mySet.add("Ibex");
mySet.add("137");
mySet.add("CS106A");
```

If you **add** a value pair where the value exists, nothing happens.




```
HashSet<String> mySet = new HashSet<String>();
mySet.add("CS106A");
mySet.add("Ibex");
mySet.add("137");
mySet.add("CS106A");

mySet.contains("Ibex");
```

Basic Set Operations

- To insert an item:
  - **set.add(value)**
- To check whether a value exists
  - **set.contains(value)**
- To remove an item
  - **set.remove(value)**

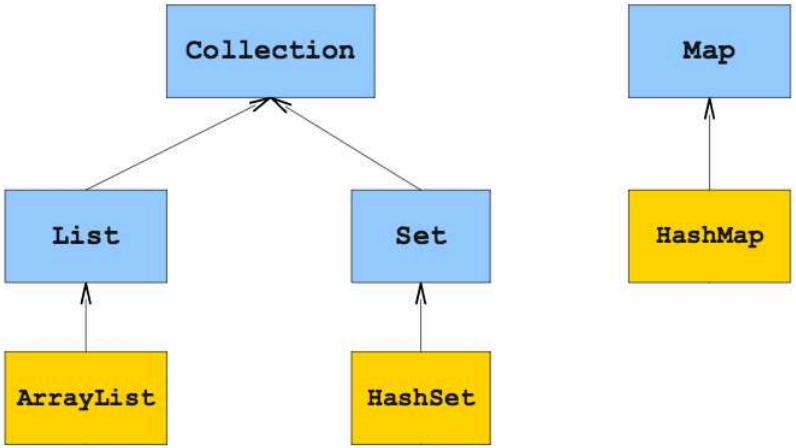
HashSet - Example

```
3   import java.util.List;
4   import java.util.Arrays;
5   import java.util.HashSet;
6   import java.util.Set;
7   import java.util.Collection;
8
9   public class SetTest
10  {
11     private static final String colors[] = { "red", "white", "blue",
12        "green", "gray", "orange", "tan", "white", "cyan",
13        "peach", "gray", "orange" };
14
15     // create and output ArrayList
16     public SetTest()
17     {
18        List< String > list = Arrays.asList( colors );
19        System.out.printf( "ArrayList: %s\n", list );
20        printNonDuplicates( list );
21     } // end SetTest constructor
```

HashSet – Example (continued)

```
    // create set from array to eliminate duplicates
    private void printNonDuplicates( Collection< String > collection )
    {
       // create a HashSet
       Set< String > set = new HashSet< String >( collection );

       System.out.println( "\nNonduplicates are: " );

       for ( String s : set )
          System.out.printf( "%s ", s );

       System.out.println();
    } // end method printNonDuplicates

    public static void main( String args[] )
    {
       new SetTest();
    } // end main
 } // end class SetTest
```

What is a List?



java.util.List interface

- A List (sometimes called a **sequence**) is an ordered Collection
- It can contain duplicate elements
- Like array indices, List indices are zero based (i.e., the first element's index is zero)
- In addition to the methods inherited from Collection, List provides methods for;
  - manipulating elements via their indices
  - manipulating a specified range of elements
  - searching for elements
  - getting a **ListIterator** to access the elements.

Iterators

- To visit every element of a collection, you can use the "for each" loop:

```
for(ElemType elem: collection){
        …
}
```

- Alternatively, you can use an **iterator**, an object whose job is to walk over the elements of a collection.
- The iterator has two commands:
  - **hasNext()** returns true if there are more items to visit.
  - **next()** returns the next item and moves the iterator to the next position.

ArrayList and Iterator - Example

```
3   import java.util.List;
4   import java.util.ArrayList;
5   import java.util.Collection;
6   import java.util.Iterator;
7
8   public class CollectionTest
9   {
10     private static final String[] colors =
11        { "MAGENTA", "RED", "WHITE", "BLUE", "CYAN" };
12     private static final String[] removeColors =
13        { "RED", "WHITE", "BLUE" };
14
15     // create ArrayList, add Colors to it and manipulate it
16     public CollectionTest()
17     {
18        List< String > list = new ArrayList< String >();
19        List< String > removeList = new ArrayList< String >();
20
21        // add elements in colors array to list
22        for ( String color : colors )
23           list.add( color );
24
```

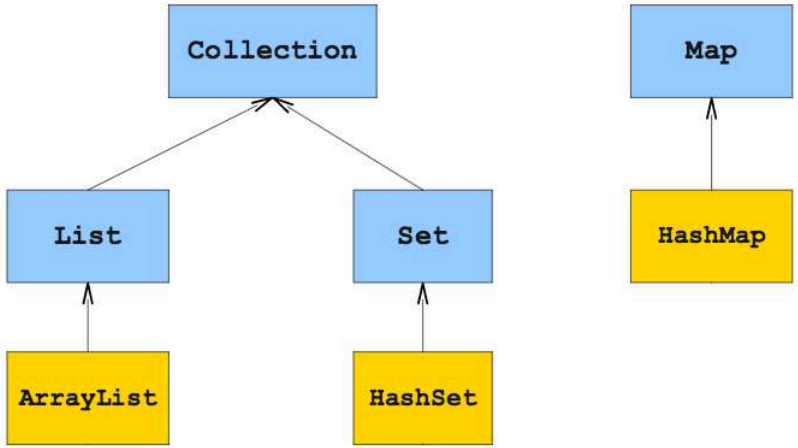
ArrayList and Iterator – Example (continued)

```
25         // add elements in removeColors to removeList
26         for ( String color : removeColors )
27            removeList.add( color );
28
29         System.out.println( "ArrayList: " );
30
31         // output list contents
32         for ( int count = 0; count < list.size(); count++ )
33            System.out.printf( "%s ", list.get( count ) );
34
35         // remove colors contained in removeList
36         removeColors( list, removeList );
37
38         System.out.println( "\n\nArrayList after calling removeColors: " );
39
40         // output list contents
41         for ( String color : list )
42            System.out.printf( "%s ", color );
43      } // end CollectionTest constructor
```

ArrayList and Iterator - Example (continued)

```
45      // remove colors specified in collection2 from collection1
46      private void removeColors(
47         Collection< String > collection1, Collection< String > collection2 )
48      {
49         // get iterator
50         Iterator< String > iterator = collection1.iterator();
51
52         // loop while collection has items
53         while ( iterator.hasNext() )
54
55            if ( collection2.contains( iterator.next() ) )
56               iterator.remove(); // remove current Color
57      } // end method removeColors
58
59      public static void main( String args[] )
60      {
61         new CollectionTest();
62      } // end main
63   } // end class CollectionTest
```

What is a Map?



java.util.Map interface

- **Map**s associate keys to values and cannot contain duplicate keys (i.e., each key can map to only one value)
- **Map**s differ from **Set**s in that **Map**s contain keys and values, whereas **Set**s contain only values

HashMap - Example

```
 3  import java.util.StringTokenizer;
 4  import java.util.Map;
 5  import java.util.HashMap;
 6  import java.util.Set;
 7  import java.util.TreeSet;
 8  import java.util.Scanner;
 9
10  public class WordTypeCount
11  {
12     private Map< String, Integer > map;
13     private Scanner scanner;
14
15     public WordTypeCount()
16     {
17        map = new HashMap< String, Integer >(); // create HashMap
18        scanner = new Scanner( System.in ); // create scanner
19        createMap(); // create map based on user input
20        displayMap(); // display map content
21     } // end WordTypeCount constructor
```

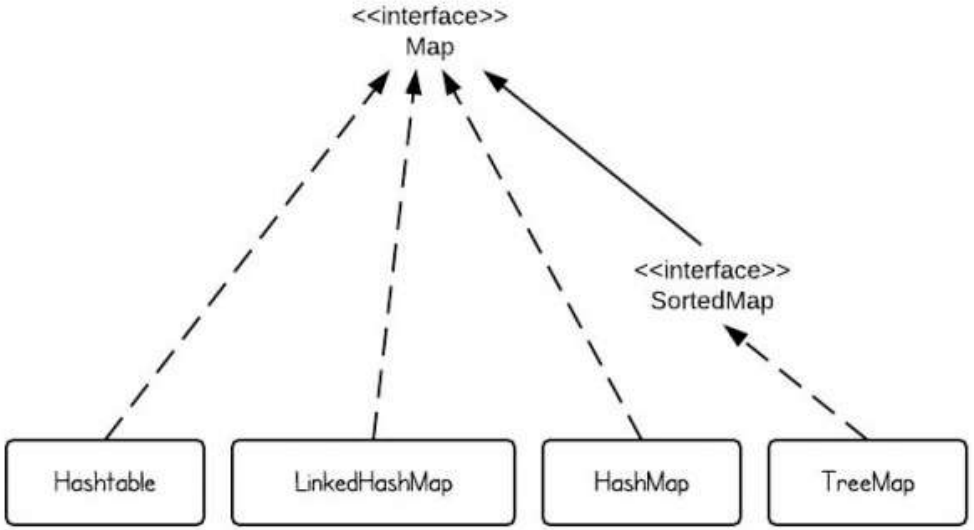
HashMap – Example (continued)

```
23      // create map from user input
24      private void createMap()
25      {
26         System.out.println( "Enter a string:" ); // prompt for user input
27         String input = scanner.nextLine();
28
29         // create StringTokenizer for input
30         StringTokenizer tokenizer = new StringTokenizer( input );
31
32         // processing input text
33         while ( tokenizer.hasMoreTokens() ) // while more input
34         {
35            String word = tokenizer.nextToken().toLowerCase(); // get word
36
37            // if the map contains the word
38            if ( map.containsKey( word ) ) // is word in map
39            {
40               int count = map.get( word ); // get current count
41               map.put( word, count + 1 ); // increment count
42            } // end if
43            else
44               map.put( word, 1 ); // add new word with a count of 1 to map
45         } // end while
46      } // end method createMap
```

HashMap – Example (continued)

```
48      // display map content
49      private void displayMap()
50      {
51         Set< String > keys = map.keySet(); // get keys
52
53         // sort keys
54         TreeSet< String > sortedKeys = new TreeSet< String >( keys );
55
56         System.out.println( "Map contains:\nKey\t\tValue" );
57
58         // generate output for each key in map
59         for ( String key : sortedKeys )
60            System.out.printf( "%-10s%10s\n", key, map.get( key ) );
61
62         System.out.printf(
63            "\nsize:%d\nisEmpty:%b\n", map.size(), map.isEmpty() );
64      } // end method displayMap
65
66      public static void main( String args[] )
67      {
68         new WordTypeCount();
69      } // end main
70   } // end class WordTypeCount
```

What is a Hashtable?



keys ⇨

Student #	Grade
107312	B+
168904	A+
...	
221655	B-

⇦ values

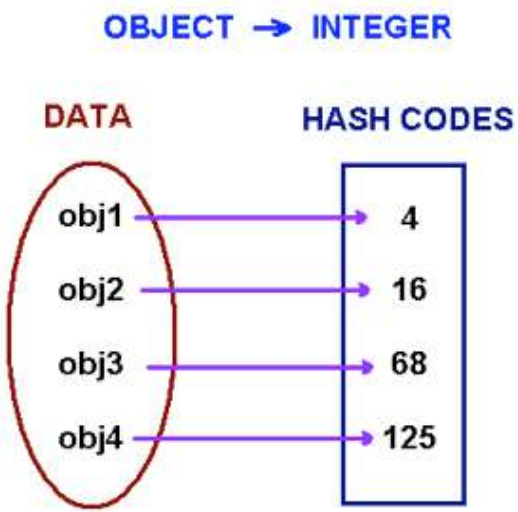
keys

Name	Ext.
Homer	1786
Marge	8113
...	
Lisa	4321

values

java.util.Properties class

- A **Properties** object is a *persistent* **Hashtable** that
  - stores key-value pairs of strings
  - assumes that you use methods **setProperty** and **getProperty** to manipulate the table rather than inherited **Hashtable** methods put and get.
  - A common use of Properties objects is to maintain application-configuration data or user preferences for applications



Properties - Example

```
3   import java.io.FileOutputStream;
4   import java.io.FileInputStream;
5   import java.io.IOException;
6   import java.util.Properties;
7   import java.util.Set;
8
9   public class PropertiesTest
10  {
11     private Properties table;
12
13     // set up GUI to test Properties table
14     public PropertiesTest()
15     {
16        table = new Properties(); // create Properties table
17
18        // set properties
19        table.setProperty( "color", "blue" );
20        table.setProperty( "width", "200" );
```

Properties – Example (continued)

```
22          System.out.println( "After setting properties" );
23          listProperties(); // display property values
24
25          // replace property value
26          table.setProperty( "color", "red" );
27
28          System.out.println( "After replacing properties" );
29          listProperties(); // display property values
30
31          saveProperties(); // save properties
32
33          table.clear(); // empty table
34
35          System.out.println( "After clearing properties" );
36          listProperties(); // display property values
37
38          loadProperties(); // load properties
```

Properties – Example (continued)

```
40          // get value of property color
41          Object value = table.getProperty( "color" );
42
43          // check if value is in table
44          if ( value != null )
45             System.out.printf( "Property color's value is %s\n", value );
46          else
47             System.out.println( "Property color is not in table" );
48       } // end PropertiesTest constructor
```

Properties – Example (continued)

```
50      // save properties to a file
51      public void saveProperties()
52      {
53         // save contents of table
54         try
55         {
56            FileOutputStream output = new FileOutputStream( "props.dat" );
57            table.store( output, "Sample Properties" ); // save properties
58            output.close();
59            System.out.println( "After saving properties" );
60            listProperties();
61         } // end try
62         catch ( IOException ioException )
63         {
64            ioException.printStackTrace();
65         } // end catch
66      } // end method saveProperties
```

Properties – Example (continued)

```
69      public void loadProperties()
70      {
71         // load contents of table
72         try
73         {
74            FileInputStream input = new FileInputStream( "props.dat" );
75            table.load( input ); // load properties
76            input.close();
77            System.out.println( "After loading properties" );
78            listProperties(); // display property values
79         } // end try
80         catch ( IOException ioException )
81         {
82            ioException.printStackTrace();
83         } // end catch
84      } // end method loadProperties
```

Properties – Example (continued)

```
86      // output property values
87      public void listProperties()
88      {
89         Set< Object > keys = table.keySet(); // get property names
90
91         // output name/value pairs
92         for ( Object key : keys )
93         {
94            System.out.printf(
95               "%s\t%s\n", key, table.getProperty( ( String ) key ) );
96         } // end for
97
98         System.out.println();
99      } // end method listProperties
100
101     public static void main( String args[] )
102     {
103        new PropertiesTest();
104     } // end main
105  } // end class PropertiesTest
```

Summary

- Arrays are used for a group of objects, but maintanence is difficult when the size of the group changes during the execution of the program
- Java Collection Framework includes many interfaces and classes to easily manage groups of objects
- The interfaces and classes of the framework are mostly in **java.util** package
- Various interfaces include methods for special algorithms of various data structures
- Framework includes various implementations of these interfaces
- It is possible to implement or extend the interfaces/classes of the framework for new/different implementations of new/different data structures

Acknowledgements

- The course material used to prepare this presentation is mostly taken/adopted from the list below:
  - Java - How to Program, Paul Deitel and Harvey Deitel, Prentice Hall, 2012
  - Building Java Programs – A Back to Basics Approach, Stuart Reges and Marty Stepp, Addison Wesley, 2011
  - Stanford, Collections – lecture notes

BBM 102 – Introduction to Programming II

Spring 2017

Exceptions

Instructors: *Ayça Tarhan, Fuat Akal, Gönenç Ercan, Vahid Garousi*

Today

- What is an exception?
- What is exception handling?
- Keywords of exception handling
  - try
  - catch
  - finally
- Throwing exceptions
  - throw
  - Custom exception classes
- Getting data from an exception object
- Checked and unchecked exceptions
  - throws

Errors

- Syntax errors
  - arise because the rules of the language have not been followed.
  - detected by the compiler.

- *Logic errors*
  - leads to wrong results and detected during testing.
  - arise because the logic coded by the programmer was not correct.

- *Runtime errors*
  - Occur when the program is running and the environment detects an operation that is impossible to carry out.

Errors

- Code errors
  - Divide by zero
  - Array out of bounds
  - Integer overflow
  - Accessing a null pointer (reference)

- Programs *crash* when an exception goes <u>untrapped</u>, i.e., not handled by the program.

Runtime Errors

```
1   import java.util.Scanner;
2
3   public class ExceptionDemo {
4     public static void main(String[] args) {
5       Scanner scanner = new Scanner(System.in);
6       System.out.print("Enter an integer: ");
7       int number = scanner.nextInt();
8
9       // Display the result
10      System.out.println(
11        "The number entered is " + number);
12    }
13  }
```

If an exception occurs on this line, the rest of the lines in the method are skipped and the program is terminated.

Terminated.

What is an exception?

- An *exception* is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

Exception = Exceptional Event

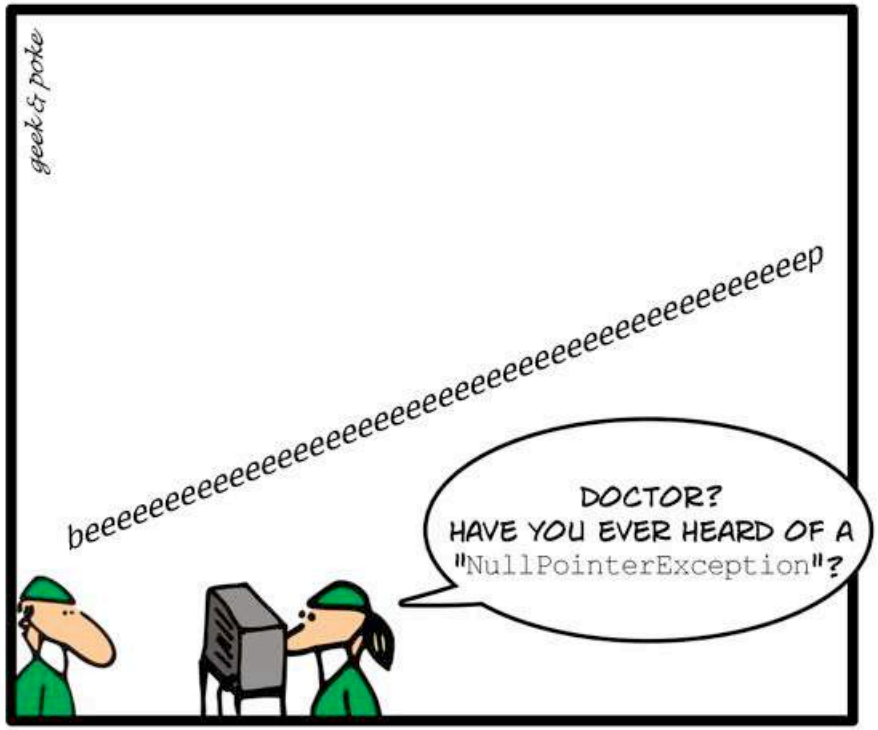
What is an exception?

- An exception is an abnormal condition that arises in a code sequence at **runtime**. For instance:
  - Dividing a number by zero
  - Accessing an element that is out of bounds of an array
  - Attempting to open a file which does not exist
- A Java exception is an object that describes an exceptional condition that has occurred in a piece of code
- When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error
- An exception can be caught to handle it or pass it on
- Exceptions can be generated by the Java run-time system, or they can be manually generated by your code

Exceptions

- A Method in Java throws exceptions to tell the calling code:

  "*Something bad happened. I failed.*"



RECENTLY IN THE OPERATING ROOM

What is an exception? (Example)

```
1- public class ExceptionExample {
2-   public static void main(String[] args) {
3-       int dividend = 5;
4-       int divisor = 0;
5-       int division = dividend / divisor;  // !!! Division by zero!
6-       System.out.println(" Result: " + division);
7-   }
8- }
```

Program "*crashes*" on the 5th line and the output is:

Exception in thread "main" java.lang.ArithmeticException: / by zero

at ExceptionExample.main(ExceptionExample.java:5)

Does the program really "crash"?

- Division by zero is an abnormal condition!
- Java run-time system cannot execute this condition normally
- Java run-time system creates an exception object for this condition and *throws* it
- This exception can be caught in order to overcome the abnormal condition and to make the program continue
- There is no exception handling code in the program, so JVM terminates the program and displays what went wrong and where it was. Remember the output:

*Exception in thread "main" java.lang.ArithmeticException: **/ by zero***

*at ExceptionExample.main(**ExceptionExample.java:5**)*

What is exception handling?

- Exception mechanism gives the programmer a chance to do something against an abnormal condition.

- Exception handling is performing an action in response to an exception.

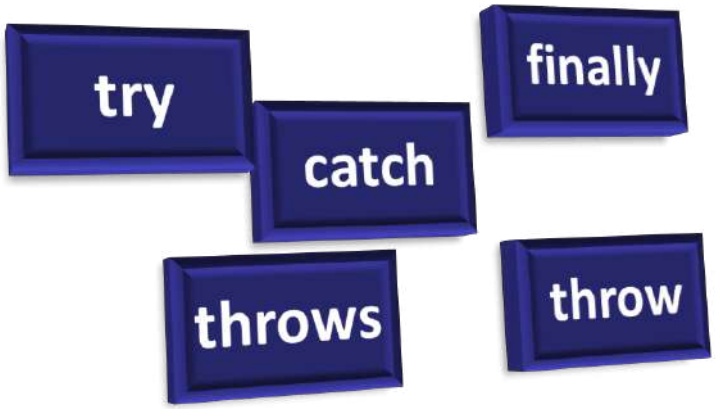
- This action may be:
  - Exiting the program
  - Retrying the action with or without alternative data
  - Displaying an error message and warning user to do something
  - ....

Keywords of Exception Handling

- There are five keywords in Java to deal with exceptions: **try, catch, throw, throws** and **finally.**

- **try**: Creates a block to monitor if any exception occurs.

- **catch**: Follows the try block and catches any exception which is thrown within it.



Let's *try* and *catch*

```
1-  public class ExceptionExample {
2-     public static void main(String[] args) {
3-       try {
4-           int dividend = 5;
5-           int divisor = 0;
6-           int division = dividend / divisor;  // !!! Division by zero!
7-           System.out.println(" Result: " + division);
8-       } catch (Exception e) {
9-           System.out.println ("Exception occurred and handled!" );
10-      }
11-    }
12- }
```

Handling Exceptions

- Java forces you to deal with checked exceptions.

- Two possible ways to deal:

```
void p1() {
  try {
    riskyMethod();
  }
  catch (IOException ex) {
    ...
  }
}
```

(a)

```
void p1() throws IOException {

  riskyMethod();

}
```

(b)

What happens when we *try* and *catch*?

- **int division = dividend / divisor;** statement causes an exception
- Java run-time system throws an exception object that includes data about the exception
- Execution stops at the 6th line, and a catch block is searched to handle the exception
- Exception is caught by the 8th line and execution continues by the 9th line
- Output of the program is:

  Exception occurred and handled!

Let's visualize it!

```
1-  public class ExceptionExample {
2-    public static void main(String[] args) {
3-      try {
4-        int dividend = 5;
5-        int divisor = 0;
6-        int division = dividend / divisor;
7-        System.out.println(" Result: " + division);
8-      } catch (Exception e) {
9-        System.out.println (" Exception occurred! " );
10-     }
11-   }
12- }
```

1. An exception is thrown by JVM

Exception object is created

e is a reference to the exception object

2. Execution stops at the exception line and diverges to the following catch block

try and catch statement

- The scope of a **catch** clause is restricted to those statements specified by the immediately preceding **try** statement.

- A **catch** statement cannot catch an exception thrown by another **try** statement.

- The statements that are protected by the **try** must be surrounded by curly braces.

Are there many exceptions in Java?

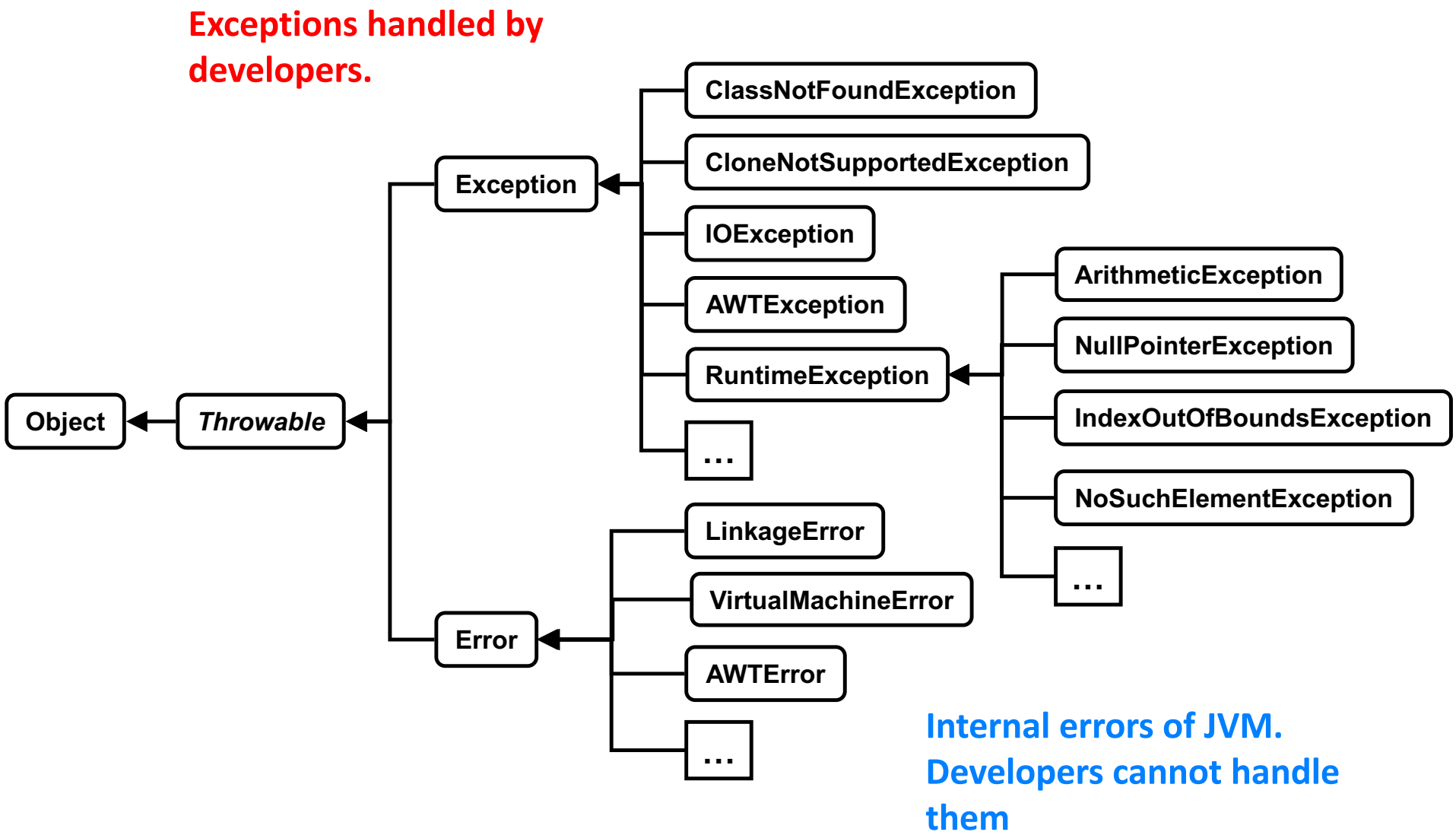
- Yes! Check the Java API Documentation at http://docs.oracle.com/javase/7/docs/api/

- `java.lang.Exception` is the base class of the exception hierarchy

- There are many direct and indirect subclasses of java.lang.Exception, for example
  - `java.lang.ArithmeticException`
  - `java.lang.ArrayIndexOutOfBoundsException`
  - `java.lang.NullPointerException`
  - `java.io.IOException`
  - `java.io.FileNotFoundException`

- We can also write custom exception classes

Hierarchy of Exception Classes in Java



Multiple catch clauses

- It is possible that more than one exception can be thrown in a code block.
  - We can use multiple **catch** clauses
- When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type *matches* that of the exception is executed.
  - Type matching means that the exception thrown must be an object of the same class or a sub-class of the declared class in the **catch** statement
- After one **catch** statement executes, the others are bypassed.

Multiple catch statement example

```
try {
    System.out.print("Give me an integer: ");
    int number = (new Scanner(System.in)).nextInt();
    System.out.println("10 / " + number + " is: " + (10 / number));
    int array[] = new int[]{1, 2, 3, 4, 5};
    System.out.println("array[" + number + "]: " + array[number]);
}
catch (ArithmeticException e) {
    System.out.println("Division by zero is not possible!");
}
catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Number is out of the array!");
}
```

ArithmeticException may occur

ArrayIndexOutOfBoundsException may occur

Multiple catch statement example

- 1st scenario: Assume that user enters value 2. What is the output of the program?

```
Give me an integer: 2
10 / 2 is: 5
array[2] is: 3
```

- 2nd scenario: Assume that user enters value 5. What is the output of the program?

```
Give me an integer: 5
10 / 5 is: 2
Number is out of the array!
```

- 3rd scenario: Assume that user enters value 0. What is the output of the program?

```
Give me an integer: 0
Division by zero is not possible!
```

Multiple catch clauses and inheritance

- If there is inheritance between the exception classes which are written in catch clauses;
  - Exception subclass must come before any of their superclasses
  - A catch statement that uses a superclass will catch exceptions of that type **plus any of its subclasses**. So, the subclass would never be reached if it comes after its superclass

```
catch (Exception e) {
}
catch (ArithmeticException e) {
}
```

Compile error! Second clause is unnecessary, because first clause will catch any exception!

```
catch (ArithmeticException e) {
}
catch (Exception e) {
}
```

It is OK now! Any exception other than an ArithmeticException will be caught by the second clause!

More on multiple catch clauses

- **Multiple catch clauses** give programmer the chance to take different actions for each exception

..., but a new catch clause for each possible exception will possibly make the code so complex

- **A single catch clause** with the java.lang.Exception will catch any exception thrown

..., but the programmer will not know which exception was thrown!

Confused about multiple catch clauses?

- Programmer decides on the details of the exception handling strategy
  - If it is just enough to know that something went wrong and the same action will be taken for all exceptions (for instance; displaying a message), then use a single catch clause with Exception!
  - If it is really necessary to know which exception occurs and different actions will be taken for each exception, then use multiple catch clauses!

Catching Exceptions

```
try {
        //Statements that may throw exceptions
}

catch (Exception1 exVar1) {
  //code to handle exceptions of type Exception1;
}

catch (Exception2 exVar2) {
  // code to handle exceptions of type Exception2;
}

...
catch (ExceptionN exVarN) {
  // code to handle exceptions of type exceptionN;
}

// statement after try-catch block
```

Catching Exceptions

An exception is thrown in

```
main method {
  ...
  try {
    ...
    invoke method1;
    statement1;
  }
  catch (Exception1 ex1) {
    //Process ex1;
  }
  statement2;
}
```

```
method1 {
  ...
  try {
    ...
    invoke method2;
    statement3;
  }
  catch (Exception2 ex2) {
    //Process ex2;
  }
  statement4;
}
```

```
method2 {
  ...
  try {
    ...
    invoke method3;
    statement5;
  }
  catch (Exception3 ex3) {
    //Process ex3;
  }
  statement6;
}
```

Nested try statements

- A **try** block can include other **try** block(s)

```
try {
    ...
    try {
        ...
    } catch (Exception e) {
        ...
    }
    ...
} catch (Exception e) {
    ...
}
```

Nested try statements

- A try block can call a method which has a try block in it.

```
try {
    ...
    method();
} catch (Exception e) {
    ...
}
```

```
void method() {
    try {
        ...
    } catch (Exception e) {
        ...
    }
}
```

Nested try statements

- When an exception occurs inside a **try** block;
  - If the **try** block does not have a matching catch, then the outer **try** statement's catch clauses are inspected for a match
  - If a matching catch is found, that catch block is executed
  - If no matching catch exists, execution flow continues to find a matching catch by inspecting the outer try statements
  - If a matching catch cannot be found, the exception will be caught by JVM's exception handler.

- Caution! Execution flow never returns to the line that exception was thrown. This means, an exception is caught and catch block is executed, the flow will continue with the lines following this catch block

Let's clarify it on various scenarios

```
try {
    statement1;
    try {
            statement2;
    } catch (Exception1 e) {
            statement3;
    } catch (Exception2 e) {
            statement4;
    }
    try {
            statement5;
    } catch (Exception3 e) {
            statement6;
    }
    statement7;
} catch (Exception3 e) {
    statement8;
}
statement9;
```

Information: Exception1 and Exception2 are subclasses of Exception3

Question: Which statements are executed if

1- statement1 throws Exception1

2- statement2 throws Exception1

3- statement2 throws Exception3

4- statement2 throws Exception1 and statement3 throws Exception2

Scenario: statement1 throws Exception1

```
try {
    statement1;
    try {
            statement2;
    } catch (Exception1 e) {
            statement3;
    } catch (Exception2 e) {
            statement4;
    }
    try {
            statement5;
    } catch (Exception3 e) {
            statement6;
    }
    statement7;
} catch (Exception3 e) {
    statement8;
}
statement9;
```

Step1: Exception is thrown

Exception1

Step2: catch clauses of the try block are inspected for a matching catch statement. Exception3 is super class of Exception1, so it matches.

Step3: statement8 is executed, exception is handled and execution flow will continue bypassing the following catch clauses

Step4: statement9 is executed

Scenario: statement2 throws Exception1

```
try {
   statement1;
   try {
            statement2;
   } catch (Exception1 e) {
            statement3;
   } catch (Exception2 e) {
            statement4;
   }
   try {
            statement5;
   } catch (Exception3 e) {
            statement6;
   }
   statement7;
} catch (Exception3 e) {
   statement8;
}
statement9;
```

Step1: Exception is thrown

Exception1

Step2: catch clauses of the try block are inspected for a matching catch statement. First clause catches the exception

Step3: statement3 is executed, exception is handled

Step4: execution flow will continue bypassing the following catch clauses. statement5 is executed.

Step5: Assuming no exception is thrown by statement5, program continues with statement7 and statement9.

Scenario: statement2 throws Exception3

```
try {
    statement1;
    try {
            statement2;
    } catch (Exception1 e) {
            statement3;
    } catch (Exception2 e) {
            statement4;
    }
    try {
            statement5;
    } catch (Exception3 e) {
            statement6;
    }
    statement7;
} catch (Exception3 e) {
    statement8;
}
statement9;
```

Step1: Exception is thrown

Exception3

Step2: catch clauses of the try block are inspected for a matching catch statement. None of these catch clauses match Exception3

Step3: Catch clauses of the outer try statement are inspected for a matching catch. Exception3 is caught and statement8 is executed

Step4: statement9 is executed

Scenario: statement2 throws Exception1 and statement3 throws Exception2

```
try {
   statement1;
   try {
            statement2;
   } catch (Exception1 e) {
            statement3;
   } catch (Exception2 e) {
            statement4;
   }
   try {
            statement5;
   } catch (Exception3 e) {
            statement6;
   }
   statement7;
} catch (Exception3 e) {
   statement8;
}
statement9;
```

Step1: Exception is thrown

Exception1

Step2: Exception is caught and statement3 is executed.

Step3: statement3 throws a new exception

Exception2

Step4: Catch clauses of the outer try statement are inspected for a matching catch. Exception2 is caught and statement8 is executed

Step5: statement9 is executed

finally

- **finally** creates a block of code that will be executed after a **try/catch** block has completed and before the following **try/catch** block
- **finally** block is executed whether or not exception is thrown
- **finally** block is executed whether or not exception is caught
- It is used to gurantee that a code block will be executed in any condition.

finally

- Use *finally* clause for code that must be executed "no matter what"

```
try {
        //Statements that may throw exceptions
}

catch (Exception1 exVar1) {
  //code to handle exceptions of type Exception1;
}

catch (Exception2 exVar2) {
  // code to handle exceptions of type Exception2;
}
...
catch (ExceptionN exVar3) {
  // code to handle exceptions of type exceptionN;
}

finally { // optional
   // code executed whether there is an exception or not
}
```

Let's clarify it on various scenarios

```
try {
        statement1;
} catch (Exception1 e) {
        statement2;
} catch (Exception2 e) {
        statement3;
} finally {
   statement4;
}
statement5;
```

Question: Which statements are executed if

1- no exception occurs

2- statement1 throws Exception1

3- statement1 throws Exception3

Scenario: no exception occurs

```
try {
        statement1;
} catch (Exception1 e) {
        statement2;
} catch (Exception2 e) {
        statement3;
} finally {
    statement4;
}
statement5;
```

Step1: statement1 is executed

Step2: finally block is executed, statement4 is executed

Step3: statement5 is executed

Scenario: statement1 throws Exception1

```
try {
        statement1;
} catch (Exception1 e) {
        statement2;
} catch (Exception2 e) {
        statement3;
} finally {
    statement4;
}
statement5;
```

Step1: Exception is thrown

Exception1

Step2: catch clauses of the try block are inspected for a matching catch statement. Exception1 is caught and statement2 is executed.

Step3: finally block is executed, statement4 is executed.

Step4: statement5 is executed

Scenario: statement1 throws Exception3

```
try {
        statement1;
} catch (Exception1 e) {
        statement2;
} catch (Exception2 e) {
        statement3;
} finally {
    statement4;
}
statement5;
```

Step1: Exception is thrown

Exception3

Step2: catch clauses of the try block are inspected for a matching catch statement. There is no matching catch. finally is executed before inspecting the outer block. statement4 is executed.

Step3: statement5 is not executed, a matching catch will be inspected at outer block(s)

throw

- Developer can *throw* exceptions. Keyword **throw** is used for this purpose:

  throw *ThrowableObject*

- *ThrowableObject* is the object to be thrown. It must directly or indirectly extend the class **java.lang.Throwable**

- Developer can create a new object of an exception class, or rethrow the caught exception

Throwing and rethrowing example

```
import java.util.Scanner;

public class ThrowingExample {
    public static void main(String[] args) {
        System.out.print("Give me an integer: ");
        int number = new Scanner(System.in).nextInt();
        try {
            if (number < 0)
                throw new RuntimeException();
            System.out.println("Thank you.");
        } catch (Exception e) {
            System.out.println("Number is less than 0!");
            throw e;
        }
    }
}
```

Keyword **throw** is used to throw an exception.

e is already reference of an exception object. It can also be used to throw (rethrow) that exception

Coding custom exception classes

- Developer can also code custom exception classes to manage abnormal conditions in his program
- If a class extends Throwable, that class can be thrown
- We usually prefer to extend class Exception or RuntimeException (difference of these two will be explained)
- Extending an exception class and coding necessary constructors is enough to create a custom exception class

Custom exception example

```
public class LessThanZeroException extends Exception {
        public LessThanZeroException() {
        }
        public LessThanZeroException(String message) {
                super(message);
        }
}
import java.util.Scanner;
public class ThrowingExample {
    public static void main(String[] args) {
        System.out.print("Give me an integer: ");
        int number = new Scanner(System.in).nextInt();
        try {
            if (number < 0)
                throw new LessThanZeroException();
            System.out.println("Thank you.");
        } catch (LessThanZeroException e) {
            System.out.println("Number is less than 0!");
        }
    }
}
```

Getting data in the exception object

- **Throwable** overrides the **toString()** method (defined by class **Object**) so that it returns a string containing a description of the exception

Example:

```
catch(ArithmeticException e) {
        System.out.println("Exception is: " + e);
}
```

Output:

Exception is: java.lang.ArithmeticException: / by zero

Getting data in the exception object

- **Throwable** class also has useful methods. One of these methods is the **getMessage()** method
- The message that is put in the exception (via the constructor with String parameter) can be taken by **getMessage()** method

Example:

```
catch(ArithmeticException e) {
        System.out.println("Problem is: " + e.getMessage());
}
```

Output:

Problem is: / by zero

Getting data in the exception object

- Another method is the **printStackTrace()** method
- This method is used to see what happened and where

Example:

```
catch(ArithmeticException e) {
        e.printStackTrace();
}
```

Output:

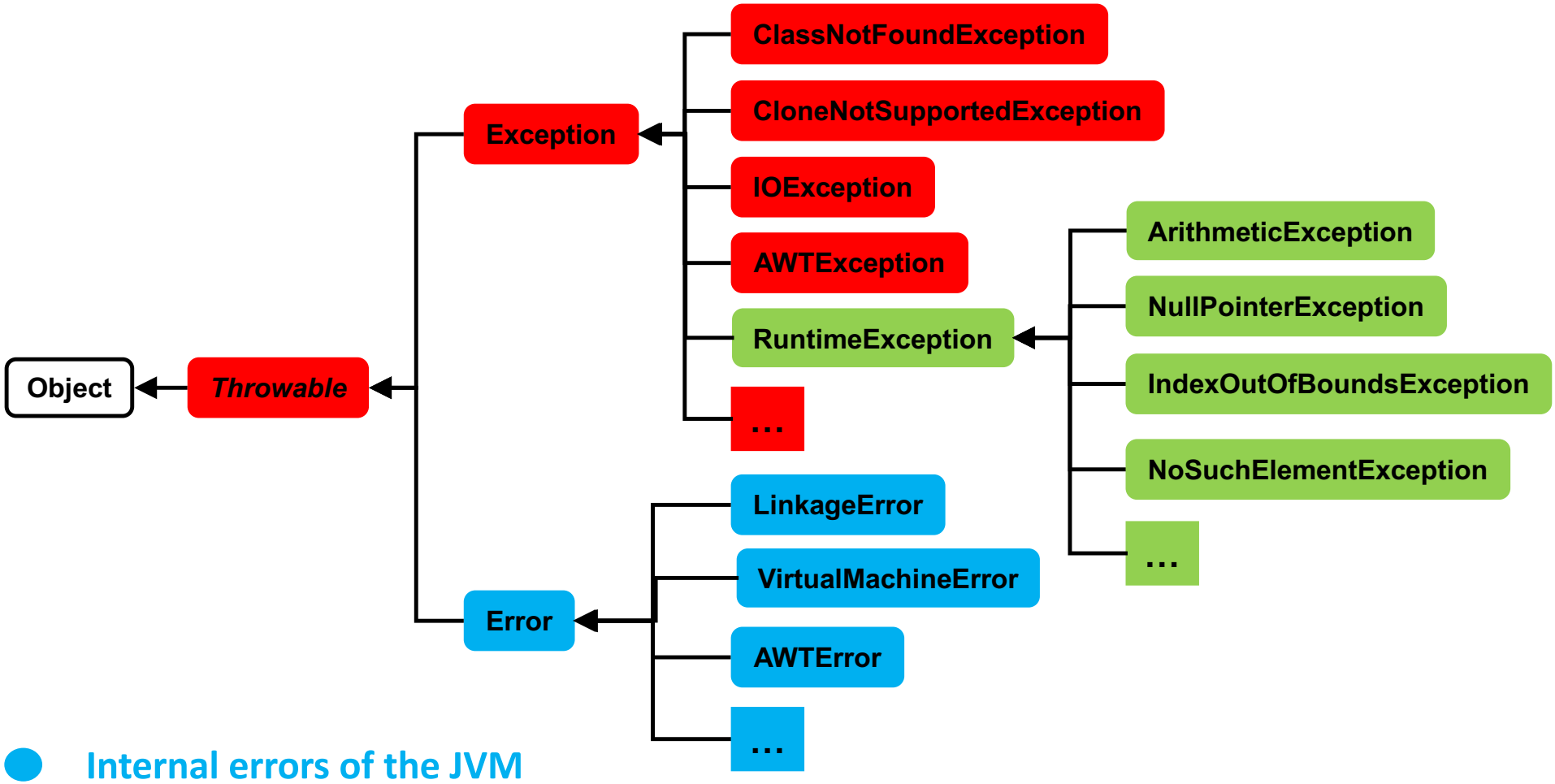
```
 java.lang.ArithmeticException: / by zero
        at ExceptionExample.main(ExceptionExample.java:6)
```

This output means:
A java.lang.ArithmeticException occurred at 6th line of the main method of the ExceptionExample class

Did you recognize that... ?

- The output of the **printStackTrace()** method is very similar to the output you have seen before...
- You have seen it when your programs crashed!
- When an exception is not caught by the program, JVM catches it and prints the stack trace to the console.
- This output is very helpful to find the errors in the program

Checked and Unchecked Exceptions



- Internal errors of the JVM
- Unchecked exceptions
- Checked exceptions

What does *Checked Exception* mean?

- If a method will possibly throw an exception, compiler *checks* the type of the exception
- if the exception is a checked exception, compiler forces the developer to do one of these:
  - write a matching catch statement for that exception
  - declare that the method will possibly throw that exception

throws

- Keyword **throws** is used to declare that a method is capable of throwing exception(s)
- Callers of the method can guard themselves against that exception(s)

Examples:

```
public void m1() throws Exception1 {
}

public void m2() throws Exception1, Exception2, Exception3 {
}
```

CheckedExceptionExample1

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class CheckedExceptionExample1 {
	public static void main(String[] args) {
		System.out.println("Line: " + readALine1("input.txt"));
	}

	public static String readALine1(String filename) {
		try {
			BufferedReader inputFile = new BufferedReader(new FileReader("a.txt"));
			String line = inputFile.readLine();
			inputFile.close();
			return line;
		} catch (IOException e) {
			e.printStackTrace();
			return null;
		}
	}
}
```

FileNotFoundException may be thrown here

IOException may be thrown here

IOException is super class of FileNotFoundException

CheckedExceptionExample2

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class CheckedExceptionExample2 {
    public static void main(String[] args) {
        try {
            System.out.println("Line: " + readALine2("input.txt"));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }


    public static String readALine2(String filename) throws IOException {
        BufferedReader inputFile = new BufferedReader(new FileReader("a.txt"));
        String line = inputFile.readLine();
        inputFile.close();
        return line;
    }
}
```

IOException is superclass of FileNotFoundException. No need to declare both.

FileNotFoundException may be thrown here

IOException may be thrown here

What does *Unchecked Exception* mean?

- If a code block has the possibility of throwing an unchecked exception, compiler does not force the developer for anything. It is up to the developer to do one of these:
  - to handle the exception
  - let the program crash

Does a developer let his program crash?

- Unchecked exceptions are usually results of the developer's mistakes.
  - For example, if a reference may normally be null, then it is developer's responsibility to check if it is null or not. NullPointerException should not occur in this scenario!
  - Letting program crash at the development phase will make the developer find such errors and potential bugs.

Summary

- Exceptions are used to take actions against abnormal conditions
- Exceptions are objects which are *thrown* by JVM or the developer's code
- There are many exception classes in standard java library, and custom exception classes can be coded
- Exception handling is *catching* an exception and taking an action against it
- Keywords **try**, **catch**, and **finally** are used for exception handling
- Exceptions are classified as unchecked (RuntimeException class and its subclasses), or checked (Throwable class and its subclasses, except Error and RuntimeException)
- If a method has the capability of throwing a checked exception, it must either handle the exception (with try/catch blocks), or declare it with keyword **throws**

References

- Ganesh Wisvanathan, CIS3023: Programming Fundamentals for CIS Majors II, University of Florida

BBM 102 – Introduction to Programming II

Spring 2017

Streams and Input/Output

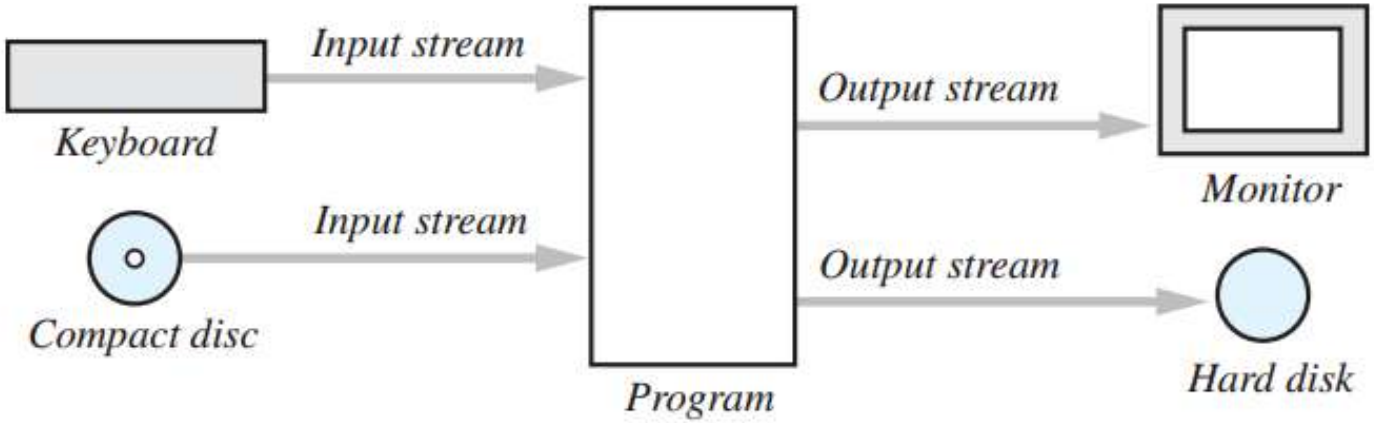
Instructors: *Ayça Tarhan, Fuat Akal, Gönenç Ercan, Vahid Garousi*

Today

- Streams and Files
- Text/Binary Files
- java.io.File class
- Revisiting java.util.Scanner
- Java I/O Library
- Decorator Pattern
- **InputStream**s and **OutputStream**s
- **Reader**s and **Writer**s
- Sequential Access vs Random Access
- java.io.RandomAccessFile
- Serialization

Streams

- A **stream** is a flow of data. The data might be characters, numbers, or bytes consisting of binary digits.
- If the data flows into your program, the stream is called an **input stream** (example: **System.in**).
- If the data flows out of your program, the stream is called an **output stream** (example: **System.out**).



Files

- The keyboard and the screen deal with temporary data
- Files provide a way to store data permanently
- All of the data in any file is stored as bits, or 0s and 1s.
- Files are categorized as **text files** and **binary files**

Text/Binary Files

- Text files
  - The bits represent printable (easily readable by humans when printed) characters.
  - The characters are coded with a "character set", ASCII, ISO-8859-1, utf-8..
  - They can be edited with a " text editor "
  - Examples: Program source files (.java, .c), files saved with a text editor, e.g. *Notepad.exe*
- Binary Files
  - The bits represent other types of encoded information, such as executable instructions or numeric data
  - They are easily read by the computer but not humans
  - They are *not* "printable" files
  - Examples: Executables (.exe), images (.jpg, .png), music (.mp3), or video (.avi, .mov) files

ASCII (American Standard Code for Information Interchange) Code Table

Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH	(start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX	(start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX	(end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT	(end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ	(enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK	(acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL	(bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS	(backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB	(horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT	(vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR	(carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO	(shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI	(shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE	(data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1	(device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2	(device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3	(device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4	(device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK	(negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN	(synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB	(end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN	(cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM	(end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB	(substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC	(escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS	(file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS	(group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS	(record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US	(unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Extended ASCII Codes

128	Ç	144	É	160	á	176	░	192	└	208	╨	224	α	240	≡
129	ü	145	æ	161	í	177	▒	193	┴	209	╤	225	ß	241	±
130	é	146	Æ	162	ó	178	▓	194	┬	210	╥	226	Γ	242	≥
131	â	147	ô	163	ú	179	│	195	├	211	╙	227	π	243	≤
132	ä	148	ö	164	ñ	180	┤	196	─	212	╘	228	Σ	244	⌠
133	à	149	ò	165	Ñ	181	╡	197	┼	213	╒	229	σ	245	⌡
134	å	150	û	166	ª	182	╢	198	╞	214	╓	230	µ	246	÷
135	ç	151	ù	167	º	183	╖	199	╟	215	╫	231	τ	247	≈
136	ê	152	ÿ	168	¿	184	╕	200	╚	216	╪	232	Φ	248	°
137	ë	153	Ö	169	⌐	185	╣	201	╔	217	┘	233	Θ	249	∙
138	è	154	Ü	170	¬	186	║	202	╩	218	┌	234	Ω	250	·
139	ï	155	¢	171	½	187	╗	203	╦	219	█	235	δ	251	√
140	î	156	£	172	¼	188	╝	204	╠	220	▄	236	∞	252	ⁿ
141	ì	157	¥	173	¡	189	╜	205	═	221	▌	237	φ	253	²
142	Ä	158	₧	174	«	190	╛	206	╬	222	▐	238	ε	254	■
143	Å	159	ƒ	175	»	191	┐	207	╧	223	▀	239	∩	255	

Source: www.LookupTables.com

www.LookupTables.com

Text/Binary Files

- Confused? Let's see an example: We want to write the number 127 into a file.

- If we write it into an ASCII coded text file:
  - Three bytes will be used for each character: 1 , 2, and 7
  - Binary values of these characters: 00110001, 00110010, 00110111

- If we write it into a binary file:
  - One byte (variable is defined as byte): 01111111
  - Two bytes (variable is defined as short): 00000000 01111111
  - Four bytes (variable is defined as int):

    00000000 00000000 00000000 01111111

java.io.File

- Do not be deceived with the name of it! Class represents a path rather than a file!
- Can be used to
  - Check if the path exists or not
  - Check if the path is a file or a directory
  - Check/edit the file/directory's readable, writable, executable, hidden properties
  - Create/delete file/directory
  - Get the contents of a directory
  - Get the last modification date and time of the file/directory

FileExample Program

```
public class FileExample {
    public static void main(String[] args) {
        File path = new File("h:\\example");
        if (!path.exists()) {                // It does not exist, create a directory!
            path.mkdir();
        } else if (path.isDirectory()) { // It is a directory! List the contents
            String[] contentOfDirectory = path.list();
            for (String filename : contentOfDirectory) {
                System.out.println(filename);
            }
        } else {                             // It is a file! Display the properties of the file
            System.out.println("Read:" + path.canRead() +
                ", Write: " + path.canWrite() + ", Hidden: " + path.isHidden());
        }
    }
}
```

Revisiting java.util.Scanner

- Class Scanner is an easy way to read input from keyboard, remember?

```
// create a scanner System.in (keyboard)
Scanner scanner = new Scanner(System.in);
// read a string from keyboard and write it to System.out (monitor)
System.out.println(scanner.next());
```

- It takes an inputstream to its constructor and reads from it

- What if we give a File object to the constructor?

```
// create a scanner for the file example.txt
scanner = new Scanner(new File("c:example.txt"));
// read a string from the file and write it to System.out (monitor)
System.out.println(scanner.next());
```

Scanner example: display contents of a file

```
public static void main(String[] args) {
        Scanner scanner = null;
        try {
                scanner = new Scanner(new File(args[0]));
                while (scanner.hasNext()) {
                        System.out.println(scanner.nextLine());
                }
        } catch (Exception e) {
                e.printStackTrace();
        } finally {
                if (scanner != null) scanner.close();
        }
}
```

Java I/O Library

- Mostly under the package **java.io**
- Includes classes, interfaces and exceptions for
  - Input/Output
  - Binary/Text
  - Sequential/Random Access
- JDK versions improved the library in time, adding new classes/interfaces.

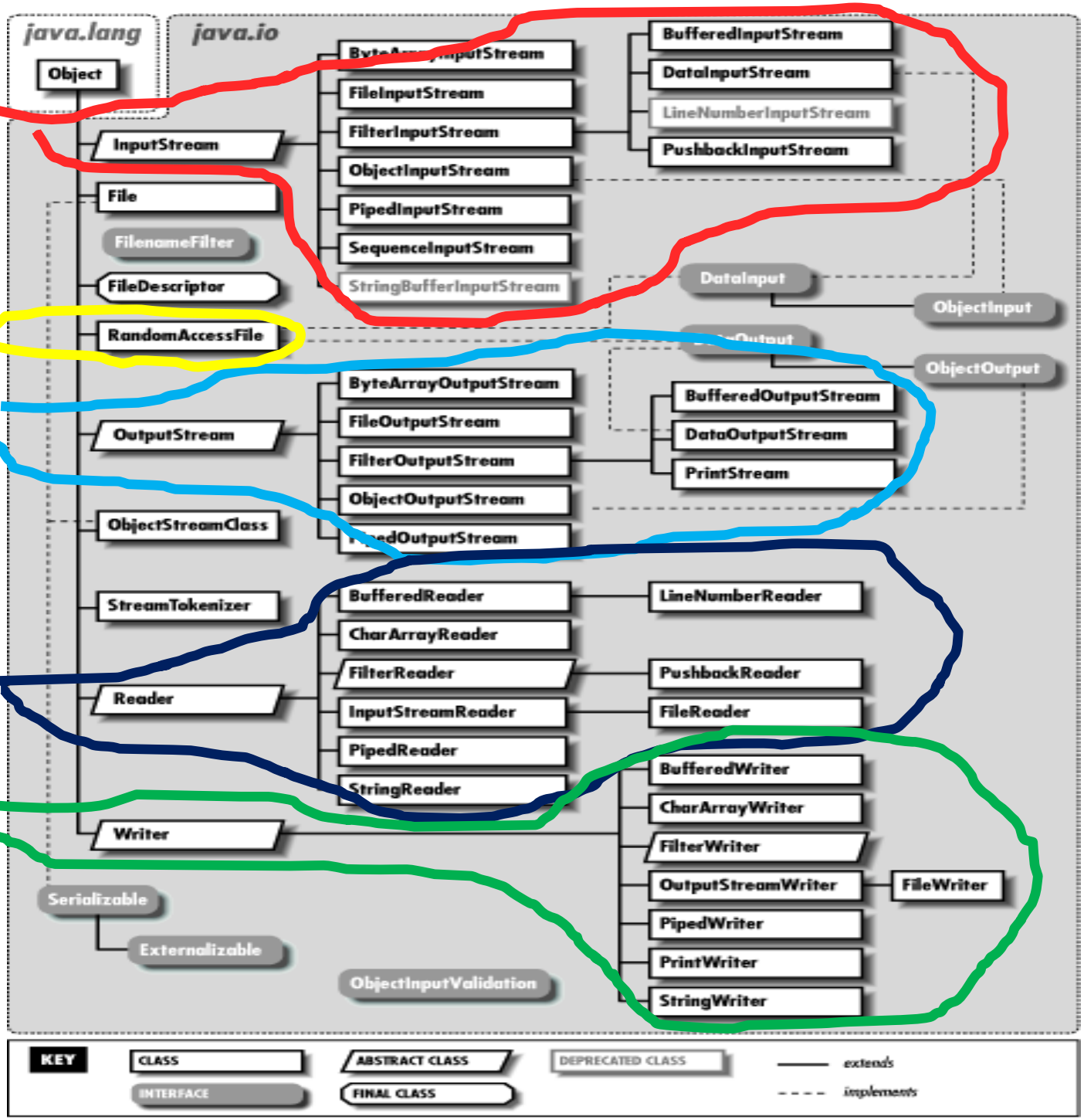
Binary Input
(byte oriented)

Random Access

Binary Output
(byte oriented)

Text Input
(character oriented)

Text Output
(character oriented)



Creating a text file

- An easy way to create a text file is using **java.io.PrintWriter**

```
public static void main(String[] args) {
    PrintWriter outputStream = null;
    try {
        outputStream = new PrintWriter("c:out.txt");  // open the file
        outputStream.println("Example line.."); // write something to the file
    } catch(FileNotFoundException e) {
        System.out.println("Error opening the file!");
    } finally {
        if (outputStream != null)    outputStream.close(); // close the file
    }
}
```

Example: from keyboard to file

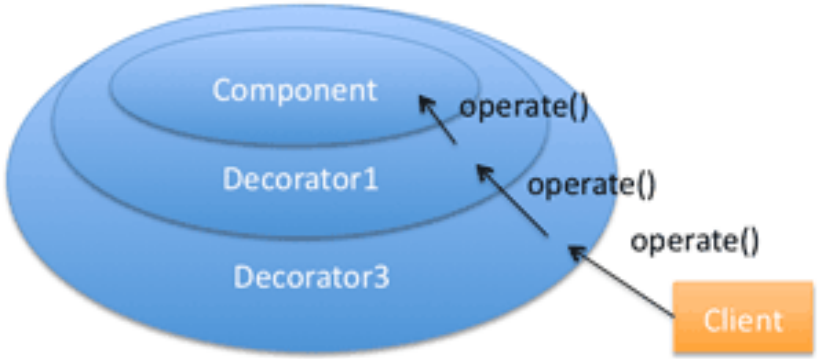
```
public static void main(String[] args) {
    PrintWriter outputStream = null;
    Scanner scanner = null;
    try {
        outputStream = new PrintWriter(args[0]);   // open the file
        scanner = new Scanner(System.in);          // create scanner for keyboard
        String str = scanner.nextLine();           // get the first line
        while (!str.equalsIgnoreCase("exit")) {    // if it is not «exit»
            outputStream.println(str);             // write it to the file
            str = scanner.nextLine();              // get a new line
        }
    } catch(FileNotFoundException e) {
        System.out.println("Error opening the file!");
    } finally {
        if (outputStream != null) outputStream.close();     // close the file
        if (scanner != null) scanner.close();               // close the scanner
    }
}
```

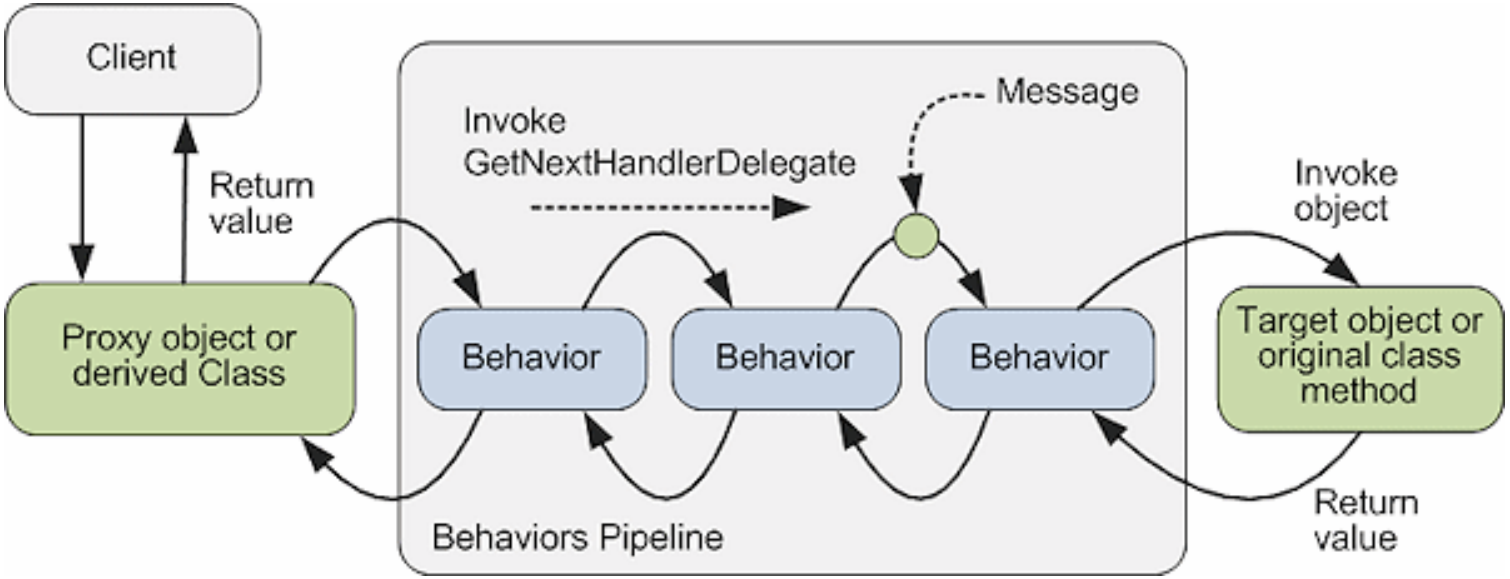
Decorator Pattern

- Software Design Patterns
  - "In software engineering, a **design pattern** is a general reusable solution to a commonly occurring problem within a given context in software design" (wikipedia)
  - Design patterns gained popularity in computer science after the book ***Design Patterns: Elements of Reusable Object-Oriented Software*** was published in 1994 by the so-called "Gang of Four" (Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides), which is frequently abbreviated as ”GoF".

Decorator Pattern

- Decorator Pattern adds a new functionality to an existing object
- A decorator class decorates an inner object and uses its methods to serve in a different way





Decorator Pattern in java.io

- **InputStream** and **Reader** classes (and their subclasses) has basic methods called **read()** for reading a single byte or an array of bytes
- **OutputStream** and **Writer** classes (and their subclasses) has basic methods called **write()** for writing a single byte or an array of bytes
- Problem: A new access to the disk for each byte will slow down the application seriously
- Solution: Bytes may be collected before reading from or writing to the disk. This will reduce the number of physical disk operations
- Decorator classes
  - java.io.BufferedInputStream, java.io.BufferedReader
  - java.io.BufferedOutputStream, java.io.BufferedWriter

BufferedReader example

```
public static void main(String[] args) {
    BufferedReader reader = null;
    try {
        reader = new BufferedReader(new FileReader(new File(args[0])));
        String line;
        while ((line = reader.readLine()) != null) {
            System.out.println(line);
        }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        if (reader != null) reader.close();
    }
}
```

A more complicated decoration example

- Let's say that we have a bunch of Java objects in a Gzipped file named ‘objects.gz’ and that we want to read them a bit quickly

```
// First open an inputstream of it:
FileInputStream fis = new FileInputStream("objects.gz");
// We want speeeed, so let's buffer it in memory:
BufferedInputStream bis = new BufferedInputStream(fis);
// The file is gzipped, so we need to ungzip it:
GzipInputStream gis = new GzipInputStream(bis);
// We need to read those Java objects:
ObjectInputStream ois = new ObjectInputStream(gis);
// Now we can finally use it:
SomeObject someObject = (SomeObject) ois.readObject();
```

InputStream and subclasses

InputStream's job is to represent classes that produce input from different sources. These sources can be:

- An array of bytes (**java.io.ByteArrayInputStream**)
- A String object (**java.io.StringBufferInputStream**)
- A file (**java.io.FileInputStream**)
- A "pipe," (**java.io.PipedInputStream**)
  - Pipe works like a physical pipe: You put things in at one end and they come out the other.
- A sequence of other streams, so you can collect them together into a single stream (**java.io.SequenceInputStream**)
- Other sources, such as an Internet connection

OutputStream and subclasses

- An array of bytes (**java.io.ByteArrayOutputStream**)
- A file (**java.io.FileOutputStream**)
- A "pipe," (**java.io.PipedOutputStream**)
  - Pipe works like a physical pipe: You put things in at one end and they come out the other.

Homework

- Go over the input and out stream classes mentioned in the previous two slides!
- Try to understand at least how they basically work.

Decorating InputStreams

- **java.io.DataInputStream**: read primitives (int, char, long, etc.) from a stream in a portable fashion.

- **java.io.BufferedInputStream**: prevents a physical read every time you want more data.

- **java.io.LineNumberInputStream**: Keeps track of line numbers in the input stream; you can call getLineNumber( ) and setLineNumber (int).
  - *This class incorrectly assumes that bytes adequately represent characters.*

- **java.io.PushbackInputStream**: Has a one-byte pushback buffer so that you can push back the last character read.

Decorating OutputStreams

- **java.io.DataOutputStream**: write primitives (int, char, long, etc.) from a stream in a portable fashion.

- **java.io.BufferedOutputStream**: prevent a physical write every time you send a piece of data.

- **java.io.PrintStream**: For producing formatted output. While DataOutputStream handles the storage of data, PrintStream handles display

Example Program: create a copy of a file

```
public static void main(String[] args) throws Exception {
    BufferedInputStream bis = null;
    BufferedOutputStream bos = null;
    try {
        bis = new BufferedInputStream(new FileInputStream(new File(args[0])));
        bos = new BufferedOutputStream(new FileOutputStream(new File(args[1])));
        byte oneByte;
        // read a byte. -1 will be returned at the end of the file.
        while ((oneByte = bis.read()) != -1) {
            bos.write(oneByte);                    //  write the byte to the output
        }
    } finally {
        if (bis != null) bis.close();              // close the streams
        if (bos != null) bos.close();
    }
}
```

Is it too slow?

```
public static void main(String[] args) throws Exception {
    BufferedInputStream bis = null;
    BufferedOutputStream bos = null;
    byte[] bytes = new byte[1024 * 16];   // bytes will be read in this by 16K chunks
    try {
        bis = new BufferedInputStream(new FileInputStream(new File(args[0])));
        bos = new BufferedOutputStream(new FileOutputStream(new File(args[1])));
        int size;
        while ((size = bis.read(bytes)) > -1) {
            bos.write(bytes);
        }
    } finally {
        if (bis != null) bis.close();
        if (bos != null) bos.close();
    }
}
```

Another example: download a web page

// please note that exception handling is not coded properly!!

```
public static void main(String[] args) throws Exception {
    URL url = new URL("http://web.cs.hacettepe.edu.tr/~bbm102/");
    BufferedInputStream bis = new BufferedInputStream(url.openStream());
    BufferedOutputStream bos = new BufferedOutputStream(
                    new FileOutputStream(new File("downloadedPage.html")));
    for (int c = bis.read(); c != -1; c = bis.read()) {
        bos.write(c);
    }
    bis.close();
    bos.close();
}
```

Readers and Writers

- **InputStream** and **OutputStream** classes provide functionality in the form of **byte oriented I/O**

- **Reader** and **Writer** were added to the library with Java 1.1. These classes provide **Unicode-compliant, character-based I/O**

- Almost all of the original Java I/O stream classes have corresponding **Reader** and **Writer** classes

InputStream/OutputStream Reader/Writer correspondings

InputStream/OutputStream	Reader/Writer
InputStream	Reader adapter: InputStreamReader
OutputStream	Writer adapter: OutputStreamWriter
FileInputStream	FileReader
FileOutputStream	FileWriter
StringBufferInputStream	StringReader
(no corresponding class)	StringWriter
ByteArrayInputStream	CharArrayReader
ByteArrayOutputStream	CharArrayWriter
PipedInputStream	PipedReader
PipedOutputStream	PipedWriter

Decorator correspondings

InputStream/OutputStream	Reader/Writer
BufferedInputStream	BufferedReader
BufferedOutputStream	BufferedWriter
PrintStream	PrintWriter
LineNumberInputStream	LineNumberReader
PushbackInputStream	PushbackReader

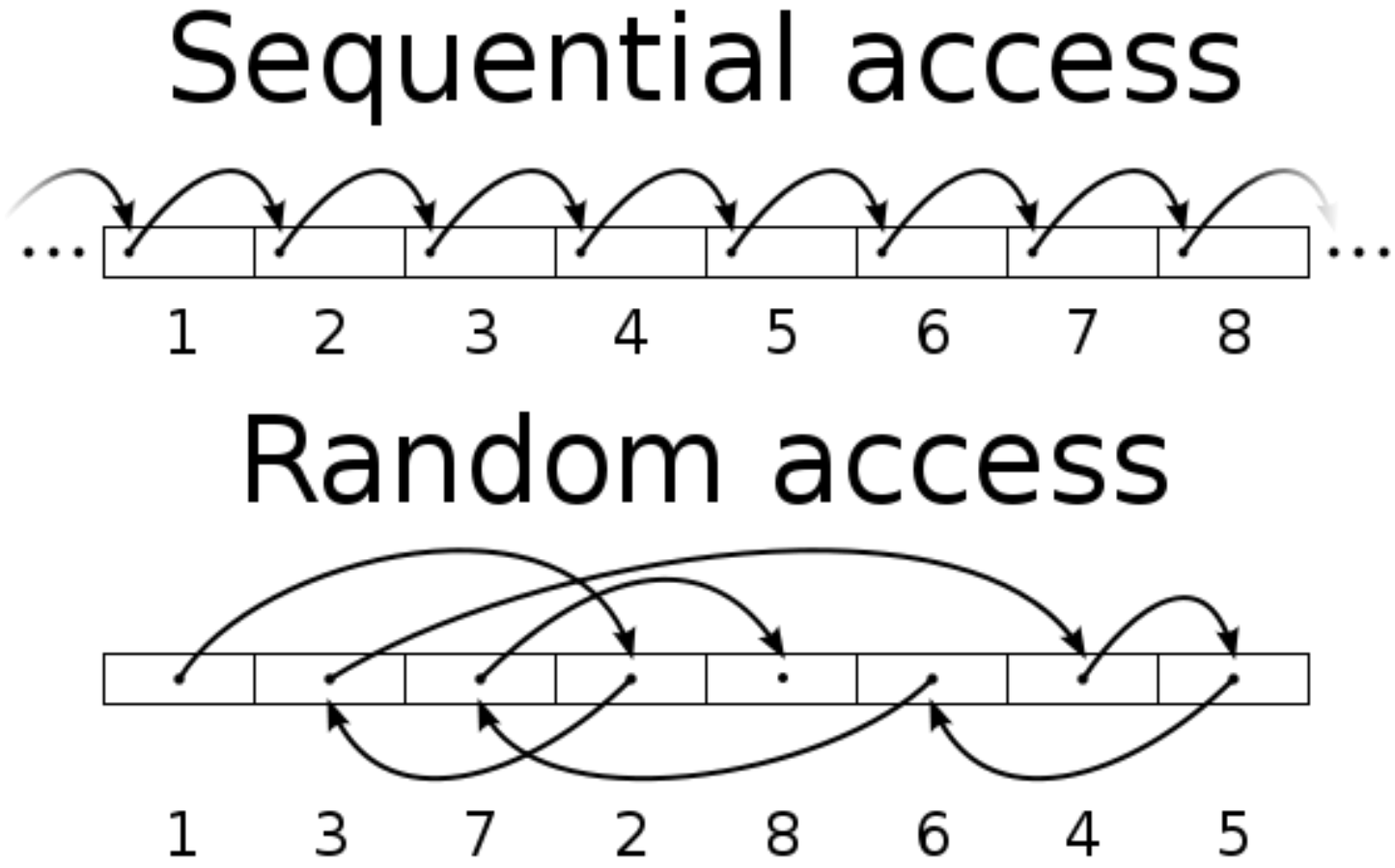
Example program: copy a file line by line

```
public static void main(String[] args) throws Exception {
    BufferedReader br = null;
    PrintWriter pw = null;
    try {
        br = new BufferedReader(new FileReader(new File(args[0])));
        pw = new PrintWriter(
                new BufferedWriter(new FileWriter(new File(args[1]))));
        String line;
        while ((line = br.readLine()) != null) {    // read a line. null at the end of the file
            pw.println(line);                       // write a line
        }
    } finally {
        if (br != null) br.close();
        if (pw != null) pw.close();
    }
}
```

Random Access

- Reading the next byte/string/number or writing to the next location is called sequential access.
- Sequential access is easy and efficient when you don’t know the contents of a file or just want to create a copy of it for example.
- On the other hand, if you know the sizes of records in a file, you can move in the file to read or change a specific record. This is random access.
- All records don’t have to be the same size; you just have to determine how big they are and where they are placed in the file.

Let’s clarify it by an image



java.io.RandomAccessFile

- Used for random access.

- Is not part of the InputStream or OutputStream hierarchy. *It's a completely separate class, written from scratch.*

- Some methods:
  - **getFilePointer( )**: find out where you are in the file
  - **seek( )**: move to a new point in the file
  - **length()**: return the length of the file

- the constructors require a second argument (*identical to fopen() in C*) indicating whether you are just randomly reading ("r") or reading and writing ("rw"). There's no support for write-only files

Example program: Editing courses Course.java

```
public class Course {
    private String code;
    private String name;
    private int credit;

    public Course(String c, String n, int cr) {
        this.setCode(c);
        this.setName(n);
        this.credit = cr;
    }
    public int getCredit() { return credit; }
    public void setCredit(int c) { this.credit = c; }

    public String getCode() { return code; }
    public void setCode(String code) {
        this.code = to40Chars(code);
    }
    public String getName() { return name; }
    public void setName(String name) {
        this.name = to40Chars(name);
    }
    private String to40Chars(String str) {
        String tmp = str;
        for (int i = str.length(); i < 40; i++) {
            tmp += ' ';
        }
        return tmp.substring(0, 40);
    }
    public String toString() {
        return code + " - " +
            name + " - " +
            credit;
    }
}
```

Program.java

RandomAccessFile is given a mode while opening the file. rw: read/write (similar to c)

```
public static void main(String[] args) throws Exception {
    // create course objects
    Course[] courses = new Course[4];
    courses[0] = new Course("BBM101", "Programlamaya Giris I", 3);
    courses[1] = new Course("BBM102", "Programlamaya Giris II", 3);
    courses[2] = new Course("BBM103", "Programlamaya Giris Lab I", 3);
    courses[3] = new Course("BBM104", "Programlamaya Giris Lab II", 3);
    // open the file. It will be accessed randomly
    RandomAccessFile raf = new RandomAccessFile(new File("courses.txt"), "rw");
    // write the courses to a file. Each course is a record
    for (int i = 0; i < courses.length; i++) {
        raf.writeBytes(courses[i].getCode());    // write the code as string
        raf.writeBytes(courses[i].getName());    // write the name as string
        raf.writeInt(courses[i].getCredit());    // write the credit as int
    }
```

Program.java (continued)

```
// let's read the second course's data and create a course object
byte[] bytes = new byte[40];          // data will be read in this as chunks of 40 bytes

// seek to the 2nd record. each record is 40 + 40 + 4 bytes long.
raf.seek((40 + 40 + 4) * (2 - 1));
```

n – 1 records must be skipped to seek to the nth record

credit

name

code

```
raf.read(bytes);
String code = new String(bytes);      // first 40 byte is the code of the course

raf.read(bytes);                      // second 40 byte is the name of the course
String name = new String(bytes);

raf.read(bytes, 0, 4);                // read 4 bytes: the credit
int credit = ByteBuffer.wrap(bytes).getInt();          // convert byte array to int
System.out.println(new Course(code, name, credit));  // create and print the course
```

Program.java

```
    // let's update the name of the 4th course

    // seek to the beginning of name of 4th course
    raf.seek((40 + 40 + 4) * (4 - 1) + 40);

    // write the new name of the course
    raf.writeBytes("Programlamaya Giris Laboratory II");

    // close the file
    raf.close();
} // end of main
```

Seek to the name field

Seek to the 4th record

Homework

- Investigate file opening modes in Java!
  - read, write, append, …

Serialization

- "**Serialization** is the process of translating data structures or object state into a format that can be stored (for example, in a file or memory buffer, or transmitted across a network connection link) and reconstructed (**deserialization**) later in the same or another computer environment" (*ref: wikipedia*)

- In Java, serialization is usually used to save/read objects to/from files using ObjectOutputStream and ObjectInputStream

- A class must implement **java.io.Serializable** interface to be serializable. It is a marker interface (has no methods to implement)

Serialization Rules in Java

- All primitive types are serializable.
- Transient fields (with transient modifier) are NOT serialized, (i.e., not saved or restored). A class that implements Serializable must mark transient fields of classes that do not support serialization (e.g., a file stream).
- Static fields (with static modifier) are Not serialized.
- If member variables of a serializable object reference to a non-serializable object, the code will compile but a RuntimeException will be thrown.

Example Program: save/read the students

```
public class Student
        implements java.io.Serializable {

      private int id;
      private String firstName;
      private String lastName;
      transient private String dummy;

      public Student(int id, String firstName,
            String lastName, String dummy) {
            this.id = id;
            this.firstName = firstName;
            this.lastName = lastName;
            this.dummy = dummy;
      }
```

```
      // getters and setters are written here

      public String toString() {
            return id + " - " +
                        firstName + " " +
                        lastName +  " " +
                        dummy;
       }

}
```

Program.java

```
public static void main(String[] args) throws Exception {
    // create students
    Student[] students = new Student[2];
    students[0] = new Student(20131234, "Ali", "Doğru", "dummy1");
    students[1] = new Student(20135678, "Veli", "Yanlış", "dummy2");
    // create the file
    ObjectOutputStream oos = new ObjectOutputStream(new
                    FileOutputStream(new File("students.dat")));
    for (int i = 0; i < students.length; i++) {
        oos.writeObject(students[i]); // write the object to file serializing
        System.out.println(students[i]);  // print the object
    }
    oos.close();            // close the file
```

Program.java (continued)

```
    // let's read and display the saved objects on the screen

    // open the file
    ObjectInputStream ois = new ObjectInputStream(
                    new FileInputStream(new File("students.dat")));
    for (int i = 0; i < students.length; i++) {
        // read the student object from file deserializing
        Student s = (Student) ois.readObject();
        System.out.println(s);
    }
    ois.close();
}
```

Output of the program

Objects written to the file:

20131234 - Ali Doğru - **dummy1**

20135678 - Veli Yanlış - **dummy2**

Objects read from the file:

20131234 - Ali Doğru - **null**

20135678 - Veli Yanlış – **null**

Note that, transient field named dummy is not serialized. So, it is null when the objects are deserialized!

java.nio.*

- Be aware of a bit more complex library of Java: The "new" I/O
- It was introduced in JDK 1.4 in the java.nio.* packages
- It’s main goal is speed. It uses ***channels*** and ***buffers*** for I/O (closer to the operating system’s way of performing I/O)
- It supports a non-blocking I/O model.

Summary

- A stream is an object that either
  - Delivers data from your program to a destination, such as a file or screen, (output stream) or
  - Takes data from a source, such as a file or the keyboard, and delivers data to your program (input stream)
- Files are handled as text or binary files
- Java has classes to handle binary (byte oriented) or text (character oriented) files
- Decoration is used to give extra functionality to existing objects. Java I/O library benefits the decoration pattern
- Java supports both sequential and random file access
- Serialization is the job of converting an object to a bit stream that can be saved or transferred to be deserialized later
- Java's nio library is a fast option for I/O

Acknowledgements

- The course material used to prepare this presentation is partially taken/adopted from the list below:
  - Thinking in Java 4th Ed., Bruce Eckel, Prentice Hall, 2006
  - Java - An Introduction to Problem Solving and Programming, Walter Savitch, Pearson, 2012