Course Overview

- What are data structures and algorithms?
- What good will it do me to know about them?
- Why can't I just use arrays and for loops to handle my data?
- What are we going to learn in this course?
  - The first course in the **Object & Structure & Algorithm** sequence
  - Object & Structure & Algorithm I focuses on **data structures**
  - Object & Structure & Algorithm II focuses on **algorithm design**
- How do we learn data structures?
- How is the grade given?

What Are Data Structures And Algorithms?

- Data structures + Algorithms = Programs
- Data structure: A specialized format for organizing and storing data
- Algorithm: A set of well-defined logical steps that must be taken to perform a task
- Any data structure is designed to organize data to suit a specific purpose so that it can be accessed and worked with in appropriate ways
- In computer programming, a data structure may be selected or designed to store data for the purpose of working on it with various algorithms

What Good Will It Do Me To Know About Them?

- Make common operations fast/efficient
- Make difficult operations possible

Common Data Structures Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

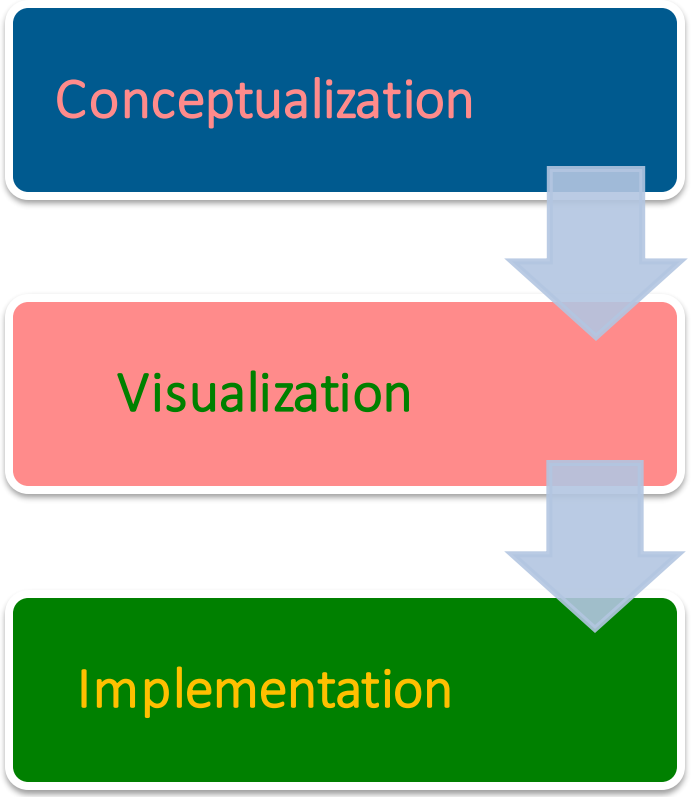
Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

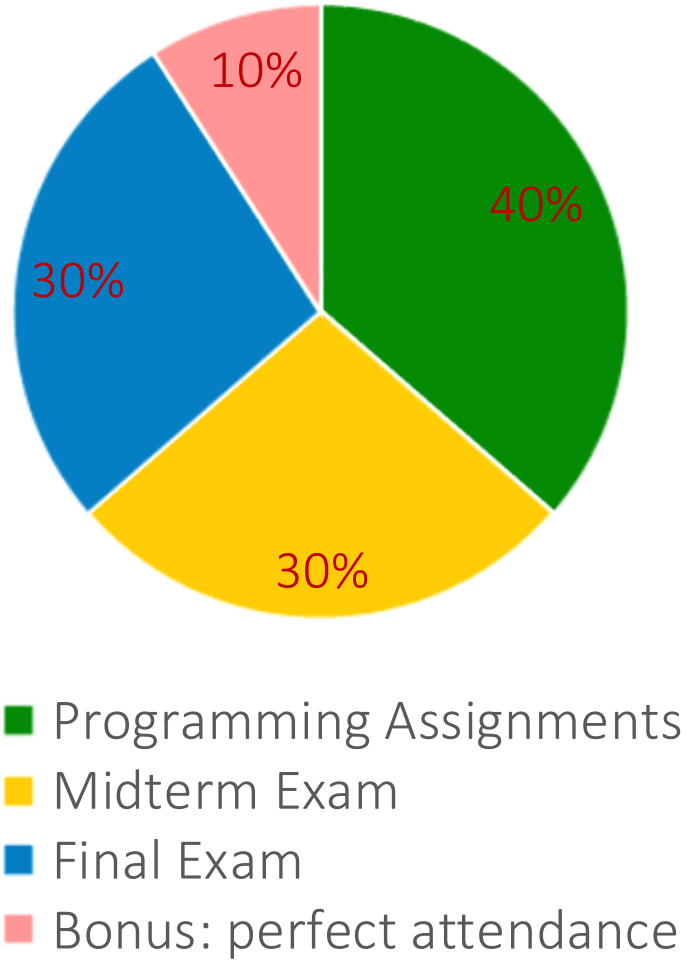
Course Outline

- Week 1 and 2: Java review
- Week 3: algorithm analysis
- Week 4: array lists and linked lists
- Week 5: array lists and linked lists
- Week 6: stacks and queues
- Week 7: sorting algorithms
- Week 8: midterm review!
- Week 9: midterm
- Week 10: binary trees
- Week 11: priority queues and heaps
- Week 12: thanksgiving
- Week 13: maps and hash tables
- Week 14: graphs
- Week 15: final review!
- Week 16: final

Learning Methods



Grades


10%
40%
30%
30%
Programming Assignments
Midterm Exam
Final Exam
Bonus: perfect attendance

FALL 2018 CISC 311

OBJECT & STRUCTURE & ALGORITHM I

– Chapter 1: Java Foundations I

Outline

- Identifier, Java keywords
- Primitive data types, casting
- Operators
- Control flow statements
- I/O methods
- Arrays

Self-test Questions

- In many data structures you can _____ a single record, _____ it, and _____ it
- Rearranging the contents of a data structure into a certain order is called _____
- In a database, a field is _____
  - a specific data item
  - a specific object
  - part of a record
  - part of an algorithm
- The field used when searching for a particular record is the _____

Self-test Questions (cont'd.)

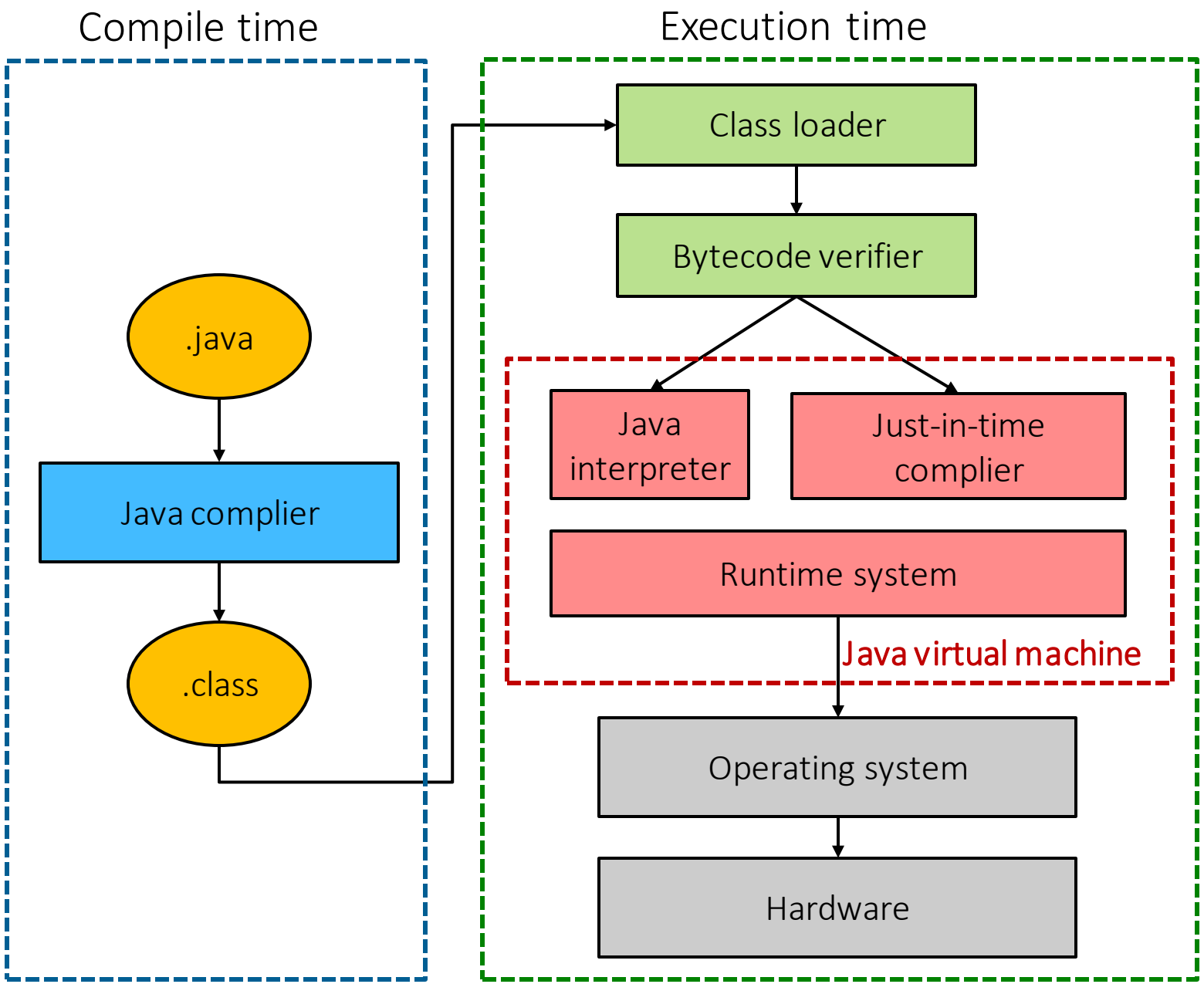
- In object-oriented programming, an object _____
  - is a class
  - may contain data and methods
  - is a program
  - may contain classes
- A class _____
  - is a blueprint for many objects
  - represents a specific real-world object
  - will hold specific values in its fields
  - specifies the type of a method

Self-test Questions (cont'd.)

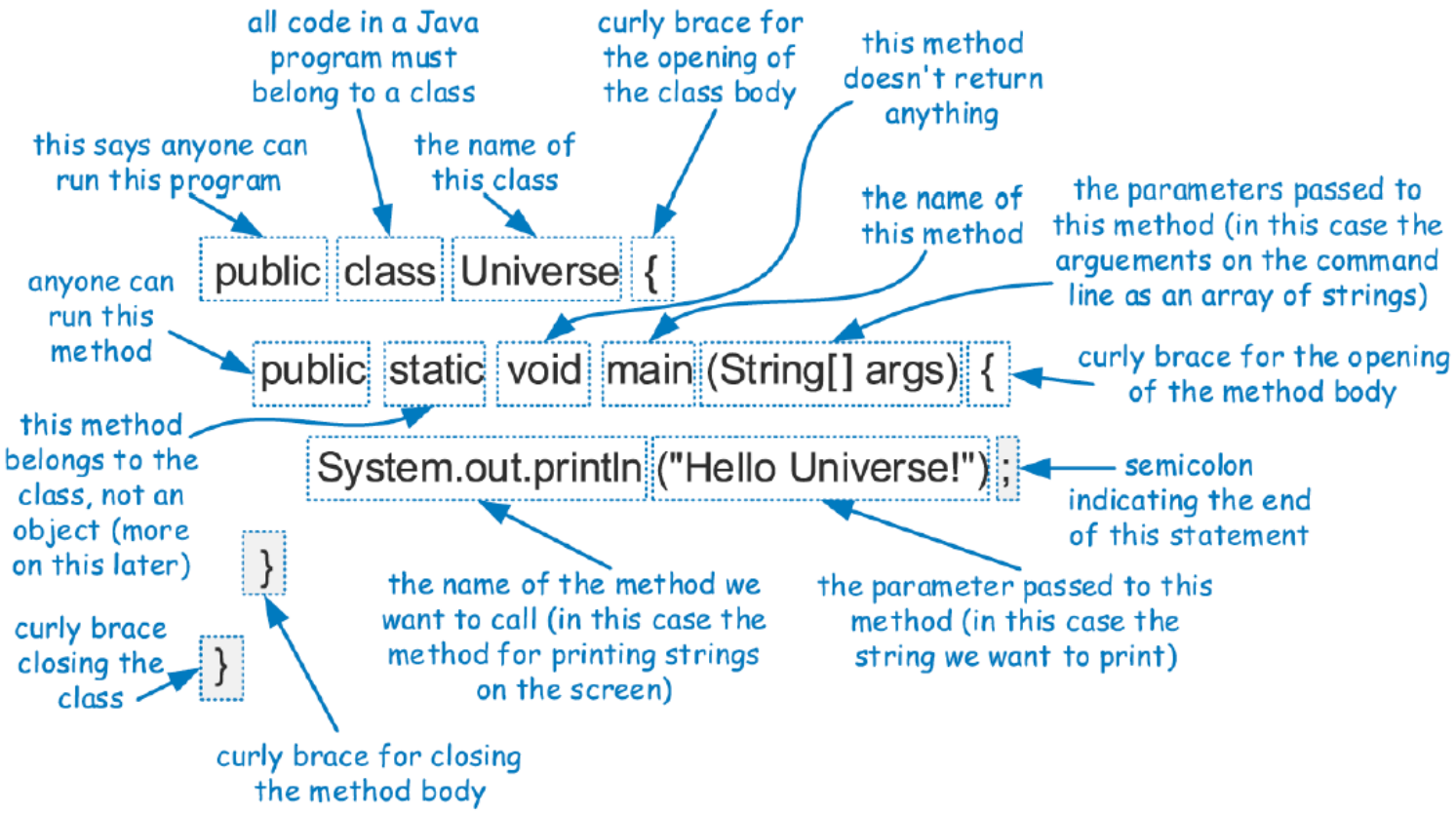
- In Java, a class specification _____
  - creates objects
  - requires the keyword new
  - creates references
  - none of the above
- When an object wants to do something, it uses a _____
- In Java, accessing an object's methods requires the _____ operator
- In Java, boolean and byte are _____

Java Complier

- Java is a compiled language
- Rather than compile straight to executable machine code, Java compiles into an intermediate binary form called JVM (Java virtual machine) byte code, which are executed through the JVM
- The JVM reads each instruction and executes that instruction
- A programmer defines a Java program in advance and saves that program in a text file known as source code
- For Java, source code is conventionally stored in a file named with the **.java** suffix (e.g., **demo.java**) and the byte-code file is stored in a file named with a **.class** suffix, which is produced by the Java compiler

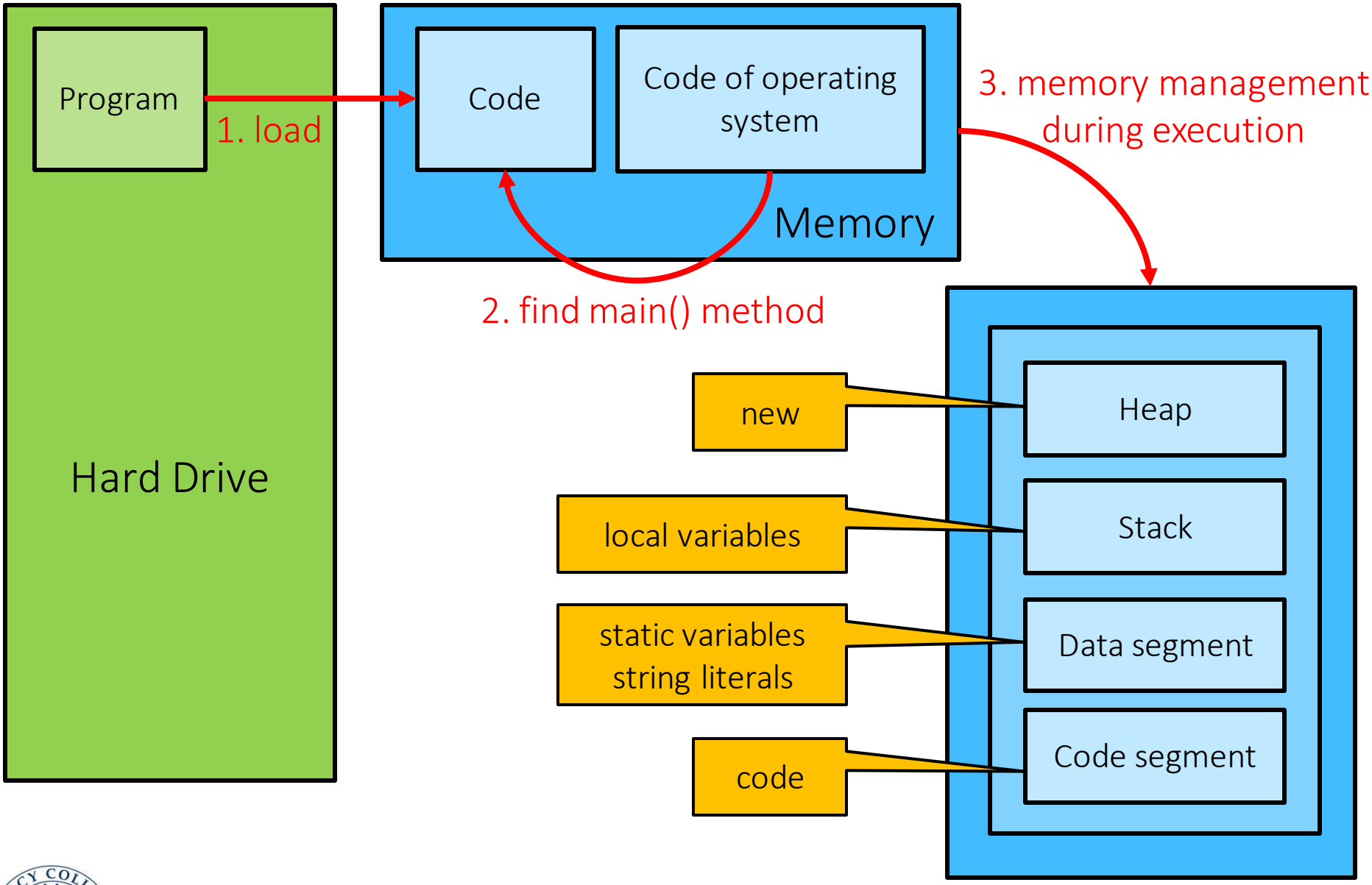

Compile time
Execution time
.java
Java complier
.class
Class loader
Bytecode verifier
Java interpreter
Just-in-time complier
Runtime system
Java virtual machine
Operating system
Hardware

An Example Program



Components of a Java Program

- In Java, executable statements are placed in functions, known as methods, that belong to class definitions
- The static method named `main` is the first method to be executed when running a Java program
- Any set of statements between the braces "{" and "}" define a program block


Program
1. load
Code
Code of operating system
Memory
3. memory management during execution
2. find main() method
Hard Drive
new
local variables
static variables
string literals
code
Heap
Stack
Data segment
Code segment

Identifiers

- The name of a class, method, or variable in Java is called an identifier, which can be any string of characters as long as it begins with a letter and consists of letters
- Exceptions: Java reserved words

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

Identifier Naming Rules

- Rules for naming identifier in Java:
  - Identifier name cannot be a key word
  - Identifier name cannot contain spaces
  - First character must be a letter, a dollar sign ($) or an underscore (_)
  - After first character may use letters, digits, dollar signs, or underscores
  - Identifier names are case sensitive
  - Identifier names are of unlimited length
- Identifier name should reflect its use
- Naming convention: camelCase
  - Readability is very important
  - Descriptive names are very useful
  - Avoid using the lowercase letter 'l', uppercase 'O', and uppercase 'I'

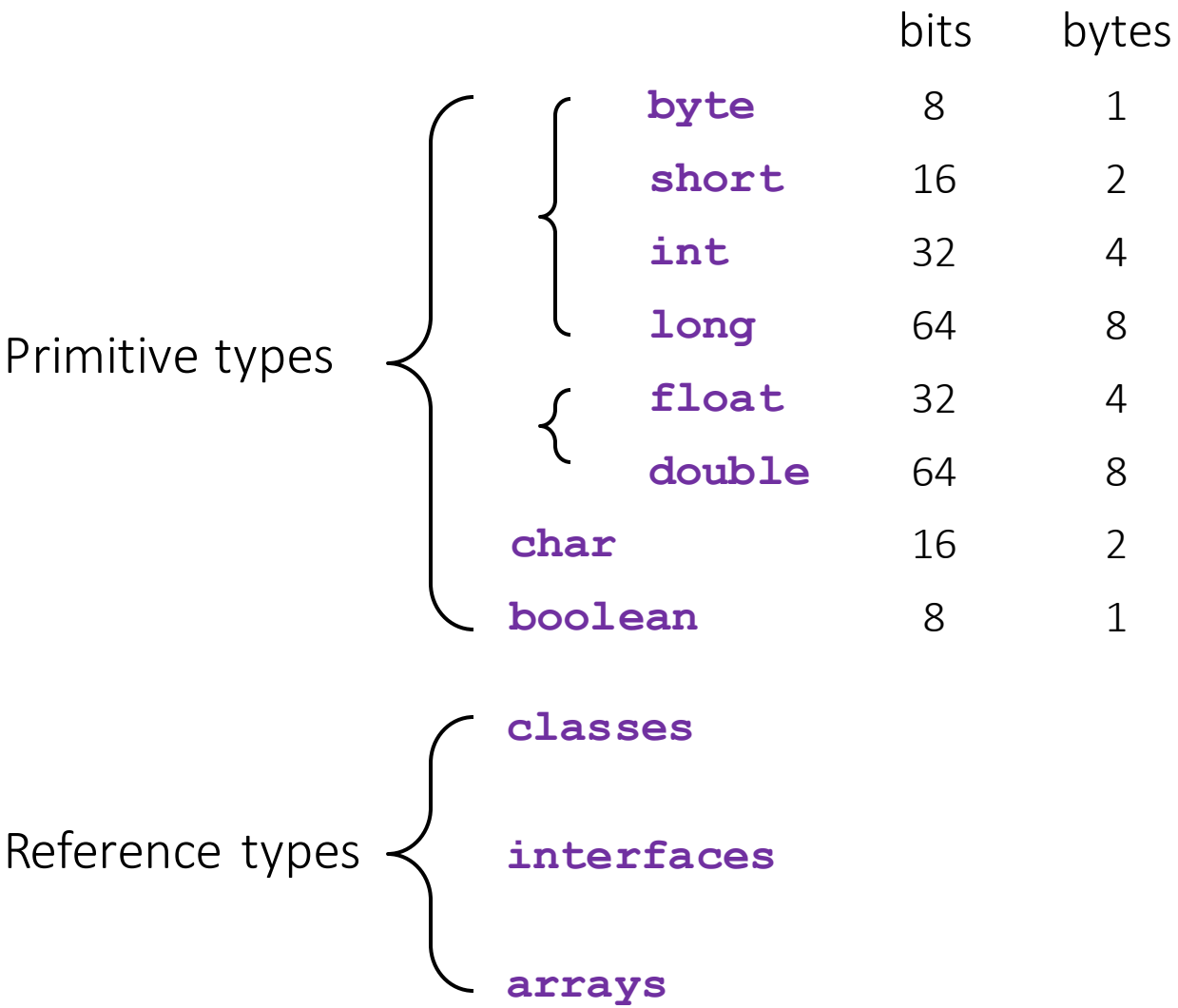
Exercise 1.1

- Legal or not?
  - `HelloWorld` ✓
  - `DataClass` ✓
  - `Class` ✓
  - `class` ✗
  - `DataClass#` ✗
  - `_983` ✓
  - `98.3` ✗
  - `$b$5_c7` ✓
  - `Hello World` ✗

Primitive Types

- Java has several base types, which are basic ways of storing data
- An identifier variable can be declared to hold any base type and it can later be reassigned to hold another value of the same type

	bits	bytes	values
`byte`	8	1	$-2^7 \sim 2^7-1$
`short`	16	2	$-2^{15} \sim 2^{15}-1$
`int`	32	4	$-2^{31} \sim 2^{31}-1$
`long`	64	8	$-2^{63} \sim 2^{63}-1$
`float`	32	4	+/- 3.4×10^{38} with 7 significant digits
`double`	64	8	+/- 1.7×10^{308} with 15 significant digits
`char`	16	2	$0 \sim 2^{16}$
`boolean`	8	1	true / false


bits
bytes
Primitive types
byte 8 1
short 16 2
int 32 4
long 64 8
float 32 4
double 64 8
char 16 2
boolean 8 1
Reference types
classes
interfaces
arrays

Casting

- Casting is an operation that allows us to change the type of a value
- We can take a value of one type and cast it into an equivalent value of another type
- There are three forms of casting in Java: explicit casting, implicit casting, and automatic casting

Explicit Casting

- Java supports an explicit casting syntax with the following form:

`(type)exp`

- Here "*type*" is the type that we would like the expression *exp* to have
- This syntax may only be used to cast from one primitive type to another primitive type, or from one reference type to another reference type
- Examples:

```
double d1 = 3.2;
double d2 = 3.999;
int i1 = (int) d1;          // i1 gets value 3
int i2 = (int) d2;          // i2 gets value 3
double d3 = (double) i2;    // d3 gets value 3.0
```

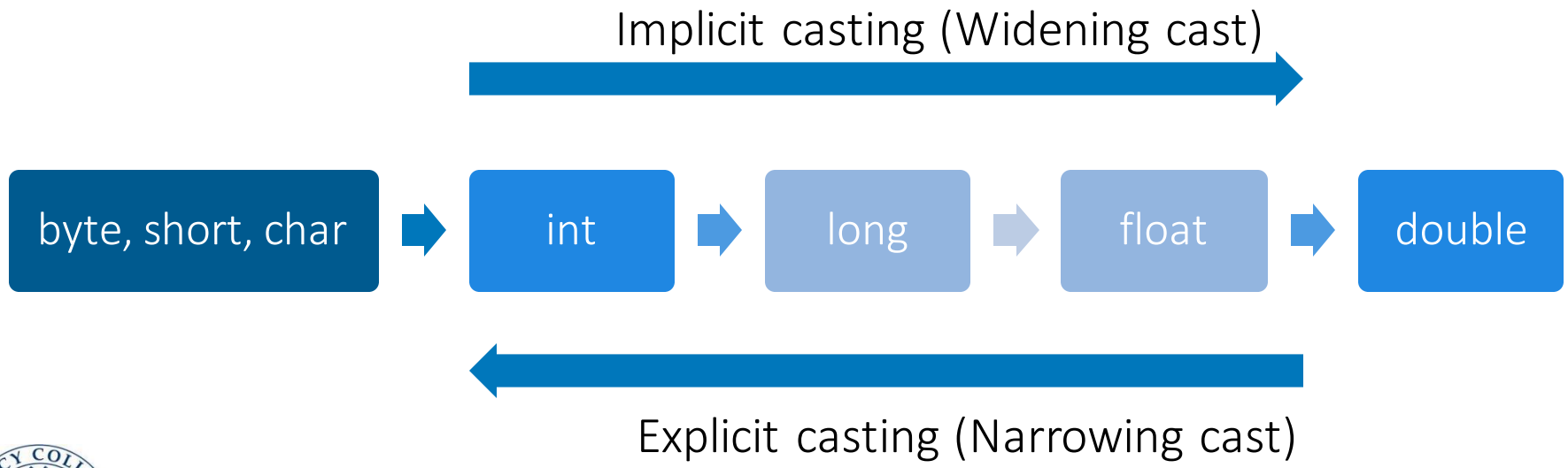
Implicit Casting

- There are cases where Java will perform an implicit cast based upon the context of an expression
- You can perform a widening cast between primitive types (such as from an int to a double), without explicit use of the casting operator
- However, if attempting to do an implicit narrowing cast, a compiler error results
- Examples:

```
int i1 = 42;
double d1  = i1; // d1 gets value 42.0
i1 = d1;  // compile error: possible loss of precision
```

Automatic Casting

- Automatic casting is similar to widening casting, it occurs implicitly between the compatible types
- An automatic type conversion will occur if the following two conditions are met:
  - The two types are compatible
  - The destination type is larger than the source type



Example

- Let's see the example `TestConversion1.java`
- Exercise 1.2:
  - Find errors or overflows in `TestCoversion2.java` and fix them

```
public class TypeConversion1 {
    public static void main(String args[]) {
        int i1 = 123, i2 = 456;
        double d1 = (i1+i2)*1.2;
        float f1 = (float)((i1+i2)*1.2);
        byte b1 = 67;
        byte b2 = 89;
        System.out.println(b1+" "+b2);
        byte b3 = (byte)(b1+b2);
        System.out.println(b3);
        double  d2 = 1e200; float f2 = (float)(d2);
        System.out.println(f2);

        float f3 = 1.23f; long l1 = 123;
        long l2 = 3000000000L;
        float f = l1 + l2 + f3;
        long i = (long)(f);
        System.out.println(f);
    }
}
```

TypeConversion1.java

```
public class TypeConversion2 {
    public static void main(String args[]) {
        int j, i = 1;
        float f1 = 0.1; float f2 = 123;
        long l1 = 12345678, l2 = 8888888888;
        double d1 = 2e20, d2 = 124;
        byte b1 = 1, b2 = 2, b3 = 129;
        j = j + 10;
        i = i / 10;
        i = i * 0.1;
        char c1 = 'a', c2 = 125;
        byte b = b1-b2;
        char c = c1+c2-1;
        float f3 = f1+f2;
        float f4 = f1+f2*0.1;
        double d = d1*i+j;
        float f = (float)(d1*5+d2);
    }
}
```

TypeConversion2.java

Wrapper Types

- There are many data structures and algorithms in Java's libraries that are specifically designed so that they only work with object types (not primitives)
- To get around this obstacle, Java defines a wrapper class for each base type
  - Java provides additional support for implicitly converting between base types and their wrapper types through a process known as automatic boxing and unboxing

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

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

Expressions and Operators

- Existing values can be combined into expressions using special symbols and keywords known as operators
- The semantics of an operator depends upon the type of its operands
  - Arithmetic operators
  - Relational operators
  - Logical operators
  - Bitwise operators
  - Assignment operators

Arithmetic Operators

- Java supports the following arithmetic operators:

Operator	Description
+	Additive operator (also used for String concatenation)
–	Subtraction operator
*	Multiplication operator
/	Division operator
%	Remainder operator

  - If both operands have type int, then the result is an int
  - if one or both operands have type float, the result is a float
  - Integer division has its result truncated

Increment and Decrement

- Java provides the plus-one increment (++) and decrement (––) operators
  - If such an operator is used in front of a variable reference, then 1 is added to (or subtracted from) the variable and its value is read into the expression
  - If it is used after a variable reference, then the value is first read and then the variable is incremented or decremented by 1
  - Examples:

```
int i = 8;
int j = i++;  // j becomes 8 and then i becomes 9
int k = ++i;  // i becomes 10 and then k becomes 10
int m = i--;  // m becomes 10 and then i becomes 9
int n = 9 + --i; //  i becomes 8 and then n becomes 17
```

```
public class IncrementOperators {
    public static void main(String args[]) {
        int i1 = 10, i2 = 20;
        int i = (i2++);
        System.out.print("i1 = "+ i);
        System.out.println(" i2 = " + i2);
        i = (++i2);
        System.out.print("i = " + i);
        System.out.println(" i2 = " + i2);
        i = (--i1);
        System.out.print("i = " + i);
        System.out.println(" i1 = " + i1);
        i = (i1--);
        System.out.print("i = " + i);
        System.out.println(" i1 = " + i1);
    }
}
```

IncrementOperators.java

Relational Operators

- Java supports the following operators for numerical values, which result in Boolean values:

Operator	Description
>	Greater than
<	Less than
==	Equal to
!=	Not equal to
>=	Greater than or equal to
<=	Less than or equal to

`.equals()` vs. ==

- `.equals()` is only used for objects comparison
- == is used to compare both primitive and objects
- String comparison is a common scenario of using both == and `equals` method. Since `java.lang.String` class override `equals` method, It return true if two String object contains same content but == will only return true if two references are pointing to the same object

```
public class TestStringComparison {
    public static void main(String args[]) {
    String s1 = new String("Comparing Strings");
    String s2 = new String("Comparing Strings");

    boolean result = (s1 == s2);
    System.out.println(result);

    result = s1.equals(s2);
    System.out.println(result);

    s1 = s2;
    result = (s1 == s2);
    System.out.println(result);
    }
}
```

TestStringComparison.java

Logical Operators

- Boolean values also have the following operators:

Operator	Description
`!`	Not
`&&`	Conditional and
`||`	Conditional or

- The `&&` and `||` operators **short circuit**, in that they do not evaluate the second operand if the result can determined based on the value of the first operand

Bitwise Operators

- Java provides the following bitwise operators for integers and booleans:

Operator	Description
&	Bitwise and
|	Bitwise or
^	Bitwise XOR
~	Bitwise compliment
<<	Left shift
>>	Right shift

a	b	!a	a&b	a|b	a^b	a&&b	a||b
true	true	false	true	true	false	true	true
true	false	false	false	true	true	false	true
false	true	true	false	true	true	false	true
false	false	true	false	false	false	false	false

```
public class LogicalOperators1 {
    public static void main(String args[]) {
        boolean a,b,c;
        a = true; b = false;
        c = a & b; System.out.println(c);
        c = a | b; System.out.println(c);
        c = a ^ b; System.out.println(c);
        c = !a; System.out.println(c);
        c = a && b; System.out.println(c);
        c = a || b; System.out.println(c);
    }
}
```

LogicalOperators1.java

Exercise 1.3

- What will be displayed if the following program is executed?

```
public class LogicalOperators2 {
    public static void main(String args[]) {
        int i = 1, j = 2;
        boolean flag1 = (i>3)&&((i+j)>5);
        boolean flag2 = (i<2)||((i+j)<6);
        System.out.println(flag1);
        System.out.println(flag2);
    }
}
```

LogicalOperators2.java

Exercise 1.4

- What will be displayed if the following program is executed?

```
public class LogicalOperators3 {
    public static void main(String args[]){
        int x = 0;
        int y = 0;
        for (int k=0; k<5; k++){
            if ((++x>2)||(++y>2)){
                x++;
            }
        }
        System.out.println(x+ " and " +y);
    }
}
```

LogicalOperators3.java

Assignment Operators

Operator	Description
=	Assignment Operator
+=	Add and assignment operator
-=	Subtract and assignment operator
*=	Multiply and assignment operator
/=	Divide and assignment operator
%=	Modulus and assignment operator
<<=	Left shift and assignment operator
>>=	Right shift and assignment operator
&=	Bitwise and assignment operator
^=	Bitwise exclusive or and assignment operator
|=	Bitwise inclusive or and assignment operator

Ternary Operator

- `?:` is a ternary operator that is part of the syntax for basic conditional expressions in several programming languages
- It is commonly referred to as the conditional operator, inline if (iif), or ternary if
- An expression *a* `?` *b* `:` *c* evaluates to *b* if the value of *a* is true, and otherwise to *c*
- Example:

```
public class TestTernary {
    public static void main(String args[]) {
        double grade = 40;
        String pass = (grade>60 ? "pass" : "fail");
        System.out.print(pass);
    }
}
```

TestTernary.java

Precedence

	. () {} ; ,
R to L	++ -- ~ ! (data type)
L to R	* / %
L to R	+ -
L to R	<< >> >>>
L to R	< > <= >= instanceof
L to R	== !=
L to R	&
L to R	^
L to R	|
L to R	&&
L to R	||
R to L	? :
R to L	= *= /= %= += -= <<= >>= >>>= &= ^= |=

High

Low

Tips: no need to memorize the precedence of operators. If you are not sure, just add ()

Flow Control Structures

- Decision structure:
  - `if`
  - `if … else`
  - `if … else if … else if … else`
  - `switch`
- Repetition structure:
  - `for`
  - `while`
  - `do … while`

`If` Statements

- The syntax of a simple `if` statement is as follows:

```
if (booleanExpression) {
    trueBody;
} else { falseBody; }
```

- *`booleanExpression`* is a boolean expression and *`trueBody`* and *`falseBody`* are each either a single statement or a block of statements enclosed in braces ("**{**" and "**}**")

Compound `if` Statements

- There is also a way to group a number of boolean tests, as follows:

```
if (firstBooleanExpression) {
    firstBody;
} else if (secondBooleanExpression) {
    secondBody;
} else if … {

} else { lastBody; }
```

`Switch` Statements

- Java provides for multiple-value control flow using the `switch` statement
- The `switch` statement evaluates an integer, string, or `enum` expression and causes control flow to jump to the code location labeled with the value of this expression
- If there is no matching label, then control flow jumps to the location labeled "`default`"
- This is the only explicit jump performed by the `switch` statement, however, so flow of control "falls through" to the next case if the code for a case is not ended with a `break` statement

`Switch` Statements (cont'd.)

```
switch (expression) {
    case firstExpression:
        firstBody;
        break;
    case secondExpression:
        secondBody;
        break;
    case …

    default:
        lastBody;
}
```

Example

- What will be displayed if the following program is executed? And how to fix it?

```
public class TestSwitch {
    public static void main(String args[]) {
        int num = 5;
        switch (num) {
            case 1:
                System.out.println("A");
            case 2:
                System.out.println("B");
            case 3:
            case 4:
            case 5:
                System.out.println("C");
            default:
                System.out.println("Error");
        }
    }
}
```

TestSwitch.java

`Break` and `Continue`

- Java supports a `break` statement that immediately terminate a `while` or `for` loop when executed within its body
- Java also supports a `continue` statement that causes the current iteration of a loop body to stop, but with subsequent passes of the loop proceeding as expected

`Break` and `Continue` (cont'd.)

- Example:

```
public class TestBreak {
    public static void main(String args[]) {
        int stop = 4;
        for (int i=1; i<=6; i++){
            if (i == stop) {
                break;
            }
            System.out.println("i = " + i);
        }
    }
}
```

`TestBreak.java`

`Break` and `Continue` (cont'd.)

- Example:

```
public class TestContinue {
    public static void main(String args[]) {
        int stop = 4;
        for (int i=1; i<=6; i++){
            if (i == stop) {
                continue;
            }
            System.out.println("i = " + i);
        }
    }
}
```

`TestContinue.java`

Exercise 1.5

- Write a program that displays the corresponding letter grade given the grading scale:

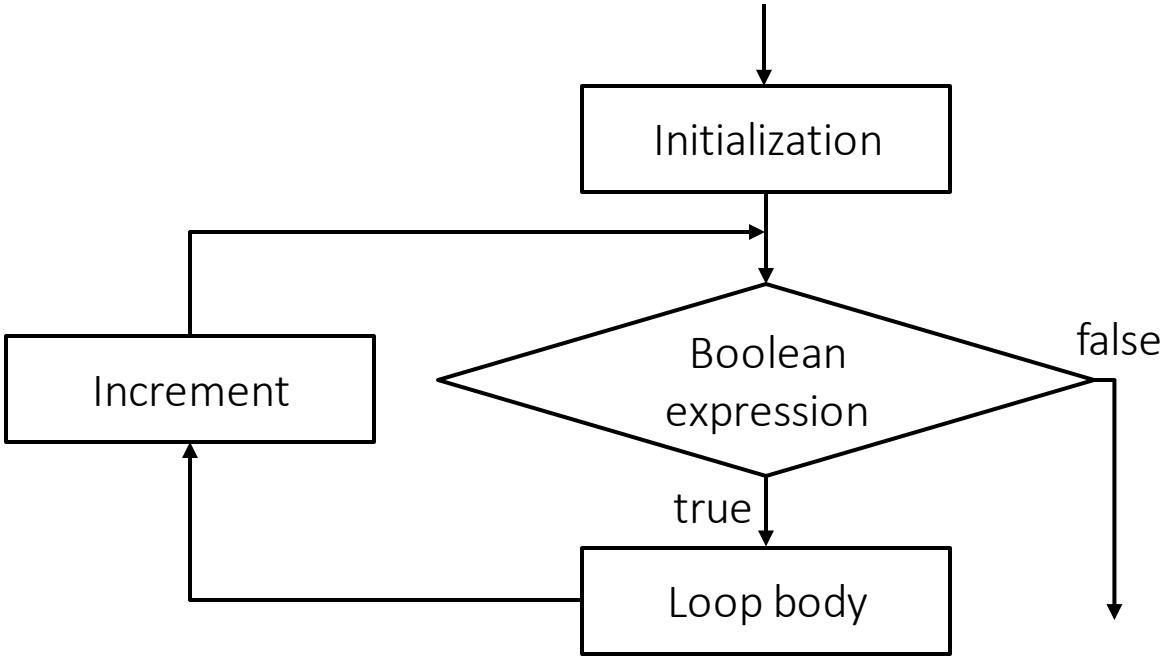
Test Score	Letter Grade
90 and above	A
80-89	B
70-79	C
60-69	D
Below 60	F
Above 100 or below 0	Invalid input

`LetterGrade.java`

`For` Loops

- The traditional `for` loop syntax consists of four sections - an initialization, a boolean condition, an increment statement, and the body - although any of those can be empty
- The structure is as follows:

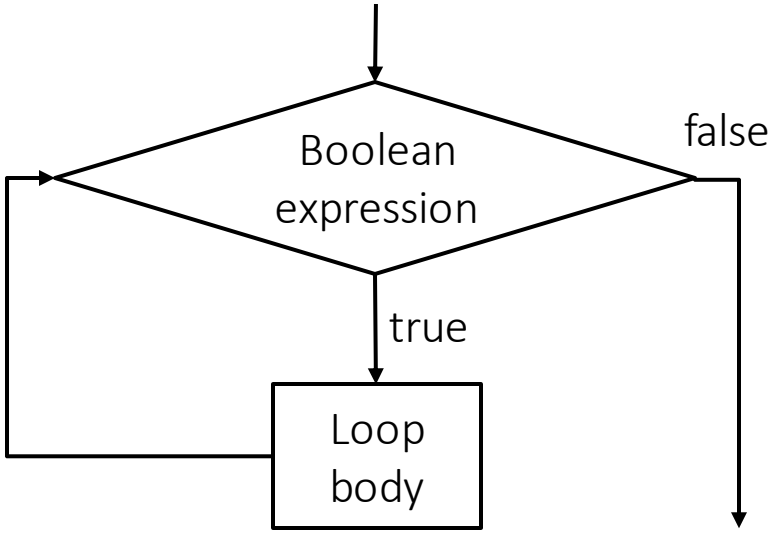
```
for (initialization; booleanExpression; increment){
        loopBody;
}
```



`While` Loops

- The simplest kind of loop in Java is a `while` loop
- Such a loop tests that a certain condition is satisfied and will perform the body of the loop each time this condition is evaluated to be true
- The syntax for such a conditional test before a loop body is executed is as follows:

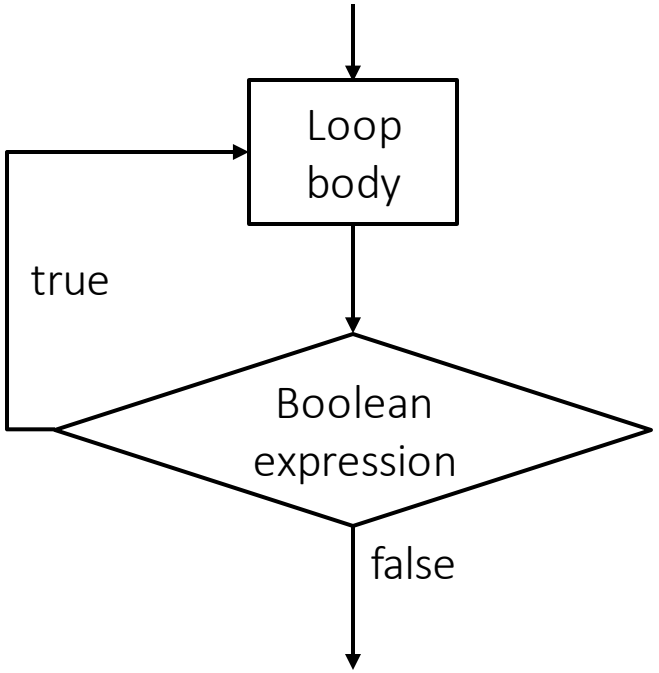
```
while (booleanExpression) {
        loopBody;
}
```



`Do-while` Loops

- Java has another form of the `while` loop that allows the boolean condition to be checked at the end of each pass of the loop rather than before each pass
- This form is known as a `do-while` loop, and has syntax shown below:

```
do {
        loopBody;
} while (booleanExpression);
```



For-Each Loops

- Since looping through elements of a collection is such a common construct, Java provides a shorthand notation for such loops, called the `for-each` loop
- The syntax for such a loop is as follows:

```
for (elementType name : container) {
        loopBody;
}
```

Exercise 1.6

- Write a program that calculates 1 + 3 + 5 + 7 + … + 99
  - By using `for` loop
  - By using `while` loop
  - By using `do-while` loop

OddSum.java

Exercise 1.7

- Ask the user to enter an integer n ($n \geq 1$).
- Write a program that calculates $1! + 2! + 3! + 4! + \ldots + n!$
  - By using `for` loop
  - By using `while` loop
  - By using `do-while` loop

FactorialSum.java

Exercise 1.8

- Write a program that displays the multiplication table shown as follows:
  - By using `for` loop
  - By using `while` loop
  - By using `do-while` loop

```
9*9=81 9*8=72 9*7=63 9*6=54 9*5=45 9*4=36 9*3=27 9*2=18 9*1=9
8*8=64 8*7=56 8*6=48 8*5=40 8*4=32 8*3=24 8*2=16 8*1=8
7*7=49 7*6=42 7*5=35 7*4=28 7*3=21 7*2=14 7*1=7
6*6=36 6*5=30 6*4=24 6*3=18 6*2=12 6*1=6
5*5=25 5*4=20 5*3=15 5*2=10 5*1=5
4*4=16 4*3=12 4*2=8  4*1=4
3*3=9  3*2=6  3*1=3
2*2=4  2*1=2
1*1=1
```

MultiplicationTable.java

Exercise 1.9

- Write a program that displays the pattern shown as follows:
  - By using `for` loop
  - By using `while` loop
  - By using `do-while` loop

```
        1
      212
    32123
   4321234
  543212345
 65432123456
7654321234567
 876543212345678
 9876543212345689
```

NumTower.java

Simple Output

- Java provides a built-in static object, called `System.out`, that performs output to the "standard output" device, with the following methods:

`print(String s)`	Print the string *s*
`print(Object o)`	Print the object *o* using its `toString` method
`print(baseType b)`	Print the base type value *b*
`println(String s)`	Print the string *s*, followed by the newline character
`println(object o)`	Similar to `print(o)`, followed by the newline character
`println(baseType b)`	Similar to `print(b)`, followed by the newline character

Simple Input

- There is also a special object, `System.in`, for performing input from the Java console window
- A simple way of reading input with this object is to use it to create a `Scanner` object, using the expression

```
new Scanner(System.in)
```

`java.util.Scanner` Methods

- The `Scanner` class reads the input stream and divides it into tokens, which are strings of characters separated by delimiters

`hasNext()`	Return `true` if there is another token in the input stream
`next()`	Return the next token string in the input stream; generate an error if there are no more tokens left
`hasNext`*`Type`*`()`	Return true if there is another token in the input stream and it can be interpreted as the corresponding base type, *Type*, where *Type* can be `boolean`, `byte`, `double`, `float`, `int`, `long`, or `short`
`next`*`Type`*`()`	Return the next token in the input stream, returned as the base type corresponding to *Type*; generate an error if there are no more tokens left or if the next token cannot be interpreted as a base type corresponding to *Type*

Example

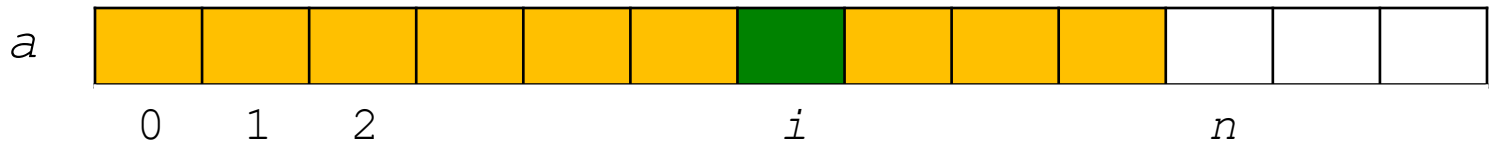
```
import java.util.Scanner;

public class TestInput {
    public static void main(String args[]) {
        Scanner input = new Scanner(System.in);
        System.out.println("Enter hours worked: ");
        int hours = input.nextInt();
        System.out.println("Enter your hourly rate: ");
        double rate = input.nextDouble();
        double salary = hours * rate;
        System.out.println("Your salary is " + salary);
    }
}
```

TestInput.java

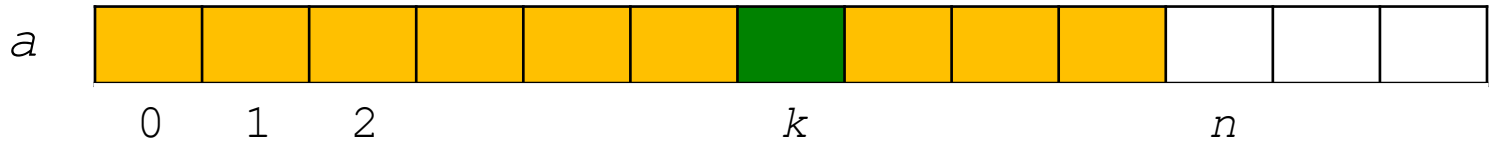
Array Definition

- An array is a sequenced collection of variables all of the same type
- Each variable, or cell, in an array has an index, which uniquely refers to the value stored in that cell
- The cells of an array, *a*, are numbered 0, 1, 2, and so on
- Each value stored in an array is often called an element of that array



Array Length and Capacity

- Since the length of an array determines the maximum number of things that can be stored in the array, we will sometimes refer to the length of an array as its capacity
- In Java, the length of an array named *a* can be accessed using the syntax `a.length`. Thus, the cells of an array, *a*, are numbered 0, 1, 2, and so on, up through `a.length-1`, and the cell with index *k* can be accessed with syntax `a[k]`



Declaring Arrays (first way)

- The first way to create an array is to use an assignment to a literal form when initially declaring the array, using a syntax as:

$$\textbf{\textit{elementType}[] \textit{arrayName} = \{\textit{initialValue}}_0\textbf{\textit{,}}$$
$$\textbf{\textit{initialValue}}_1\textbf{\textit{, …, initialValue}}_{n-1}\textbf{\textit{\};}}$$

- The *elementType* can be any Java base type or class name, and *arrayName* can be any valid Java identifier. The initial values must be of the same type as the array

Declaring Arrays (second way)

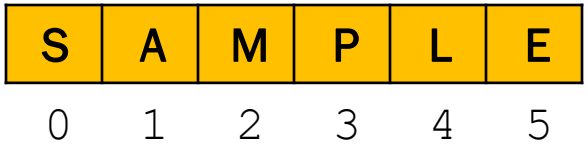
- The second way to create an array is to use the new operator
  - However, because an array is not an instance of a class, we do not use a typical constructor. Instead we use the syntax:

```
new elementType[length]
```

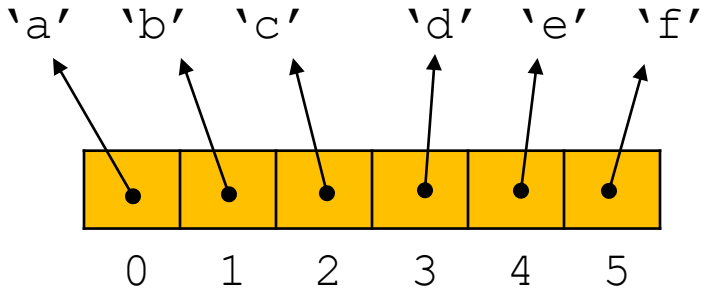
  - *length* is a positive integer denoting the length of the new array
  - The new operator returns a reference to the new array, and typically this would be assigned to an array variable

Arrays of Characters or Object References

- An array can store primitive elements, such as `char`



- An array can also store references to objects



Array Operations

- Access
- Search
- Insertion
- Deletion
- Sorting

Example

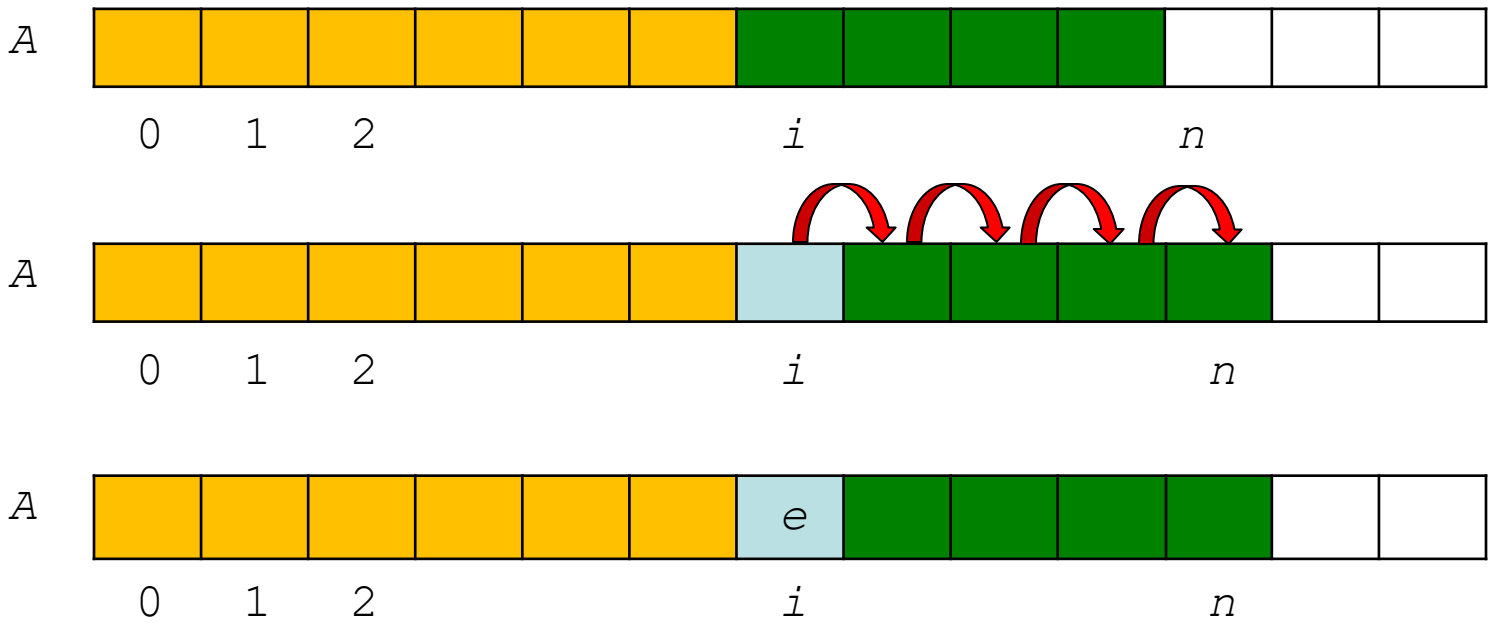
- Given an array *a* = {23, 44, 53, 21, 74, 99, 25, 45, 73, 38}, find the item with search key = 25

```
void search (int[] a, int key) {
    int i = 0;
    for (; i < a.length; i++){
        if (a[i] == key){
        break;
        }
    }
    if ( i == a.length){
        System.out.println("Unable to find " + key);
    }
    else {
        System.out.println("Find " + key);
    }
}
```

ArrayOperations.java

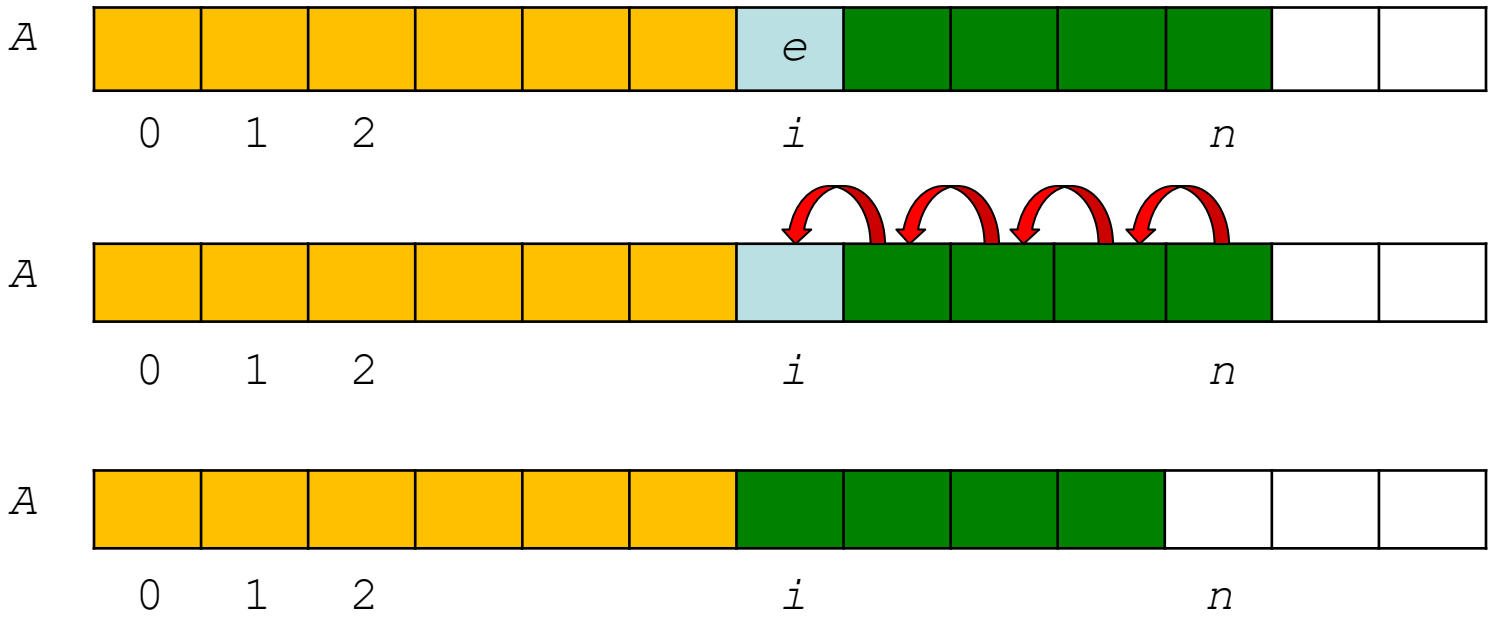
Adding an Entry: Insertion

- To add an entry e into array A at index i, we need to make room for it by shifting forward the $n-i$ entries $A[i], \ldots, A[n-1]$



Removing an Entry: Deletion

- To remove the entry e at index i, we need to fill the hole left by e by shifting backward the $n-i-1$ elements $A[i+1], \ldots, A[n-1]$



Example

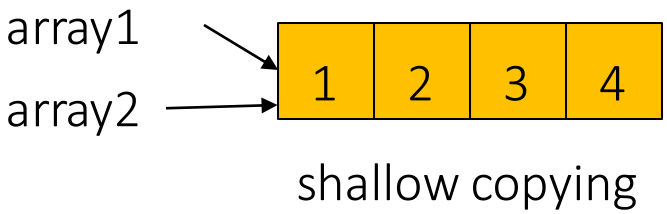
- Given an array *a* = {23, 44, 53, 21, 74, 99, 25, 45, 73, 38}, find and delete the item with search key = 74

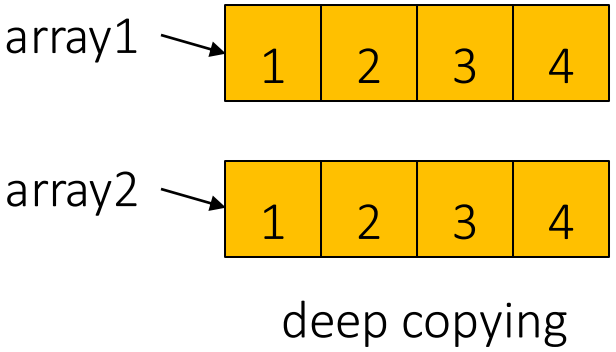
```
void delete (int[] a, int key){
    int i = 0;
    for (; i < a.length; i++){
        if (a[i] == key){
                break;
        }
    }
    for (int j = i; j < a.length-1; j++){
        a[j] = a[j+1];
    }
    System.out.println(Arrays.toString(a));
}
```

ArrayOperations.java

Copying Array: Shallow Copying and Deep Copying

- The difference between shallow and deep copying is only relevant for compound objects (objects that contain other objects, like lists or class instances):
  - A *shallow copy* constructs a new compound object and then (to the extent possible) inserts *references* into it to the objects found in the original
  - A *deep copy* constructs a new compound object and then, recursively, inserts *copies* into it of the objects found in the original





Shallow Copying

- Why array `data`'s value changes as well?

```
import java.util.Arrays;

public class ArrayShallowCopying {
    public static void main(String[] args) {
        int[] vals = {-5, 12, 0};
        int[] data;
        data = vals;
        System.out.println( Arrays.toString(vals) );
        System.out.println( Arrays.toString(data) );
        vals[0] = 13;
        System.out.println( Arrays.toString(vals) );
        System.out.println( Arrays.toString(data) );
    }
}
```

ArrayShallowCopying.java

Deep Copying

```
public static void arrayCopying1 () {
    int[] a = new int[10];
    int[] b = new int[a.length];
    for (int i = 0; i < a.length; i++){
        a[i] = i + 1;
        b[i] = a[i];
    }
    System.out.println(Arrays.toString(a));
    System.out.println(Arrays.toString(b));
    for (int i = 0; i < a.length; i++){
        a[i] = 0;
    }
    System.out.println(Arrays.toString(a));
    System.out.println(Arrays.toString(b));
}
```

ArrayCopying.java

Deep Copying (cont'd.)

```
public static void arrayCopying2 (){
    int[] a = new int[10];
    int[] b = new int[a.length];
    for (int i = 0; i < a.length; i++){
        a[i] = i + 1;
    }

    System.arraycopy(a, 0, b, 0, 10);
    System.out.println(Arrays.toString(a));
    System.out.println(Arrays.toString(b));
    for (int i = 0; i < a.length; i++){
        a[i] = 0;
    }
    System.out.println(Arrays.toString(a));
    System.out.println(Arrays.toString(b));
}
```

ArrayCopying.java

Array Sorting

- Given an array A = {1, 3, 2, 5, 6, 7, 4}, write a program that sorts numbers in the array in ascending order

```
Algorithm arraySorting1 (A)
    Input A array A
    Output A sorted array
    for i ← 0 to A.length – 2 do
        for j ← i + 1 to A.length – 1 do
            if A[j] < A[i] then
                A[j] ↔ A[i]
    return A
```

- Other solutions?

ArraySorting.java

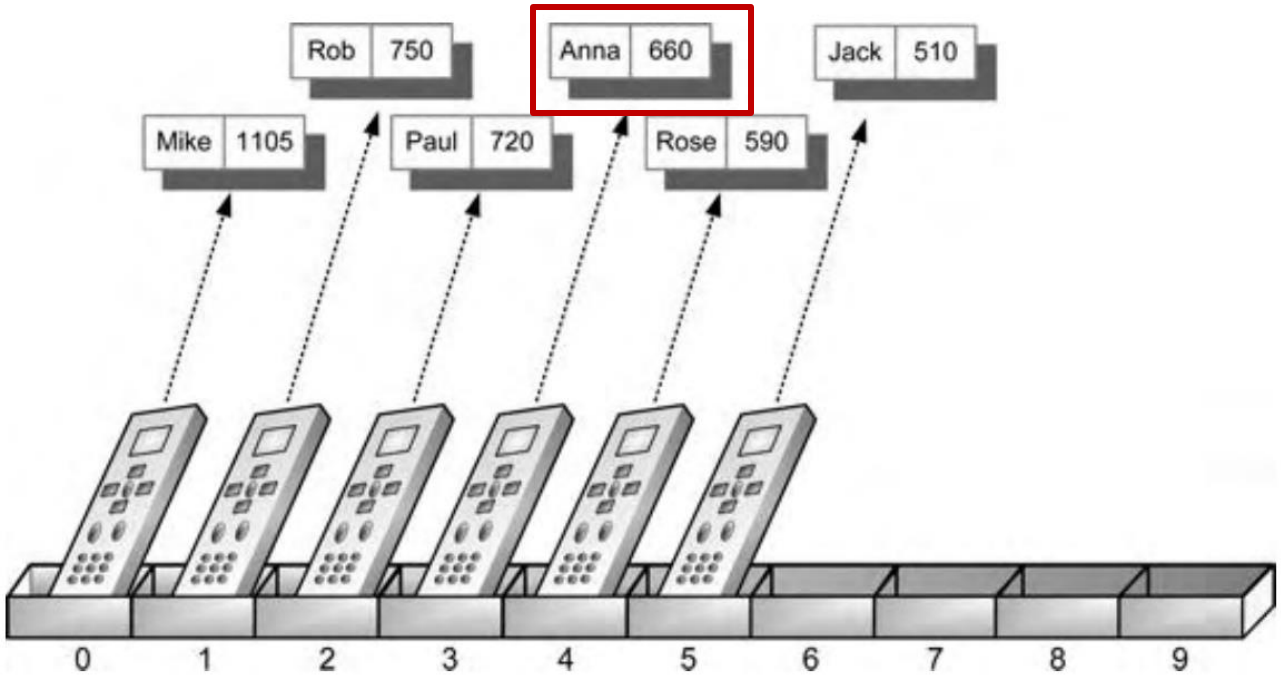
Exercise 1.10

- Given an array of doubles
  - Compute the sum of the array
  - Compute the maximum in the array

ArraySumMax.java

Exercise 1.11 : Scoreboard

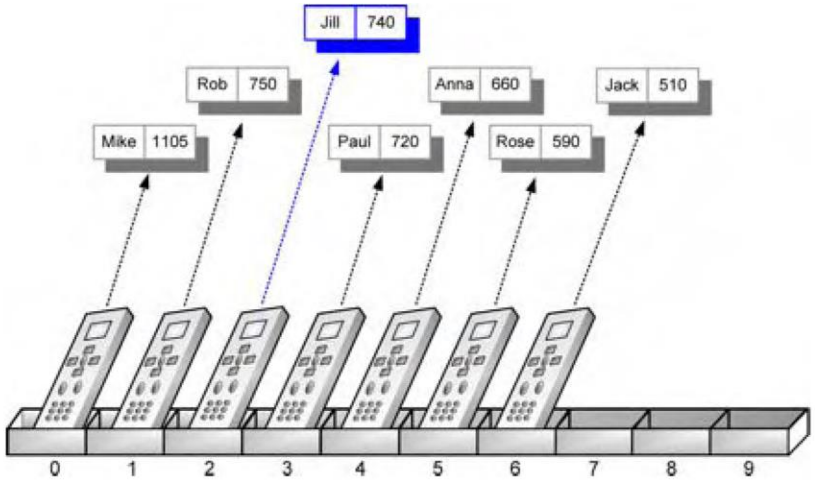
- A game entry stores the name of a player and her best score so far in a game



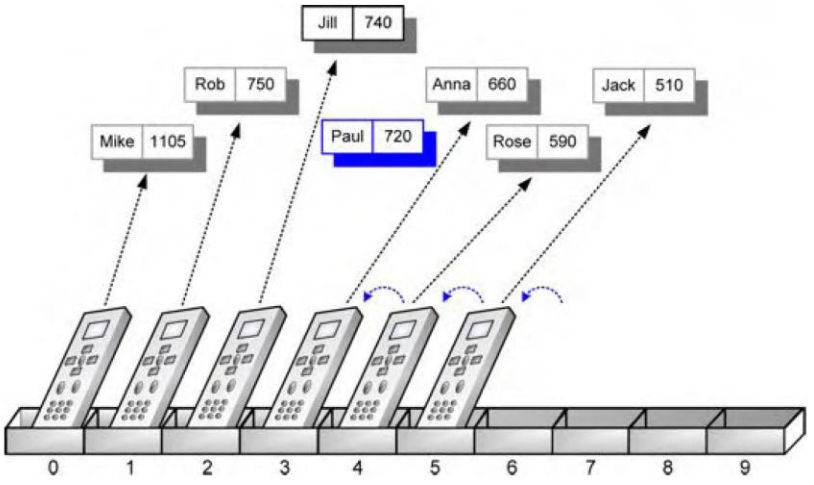
`GameEntry.java`

Example: Scoreboard (cont'd.)

- Keep track of players and their best scores in an array, board
  - The elements of board are objects of class `GameEntry`
  - Array board is sorted by score



Insertion



Deletion

`Scoreboard.java`

Summary

- A data structure is the organization of data in a computer's memory or in a disk file
- Some data structures are used as programmer's tools: they help execute an algorithm
- The correct choice of data structure allows major improvements in program efficiency
- Examples of data structures are arrays, stacks, queues, linked lists, graphs, and trees
- An algorithm is a procedure for carrying out a particular task