

600.107 Intro Programming (JAVA)

Course Webpage

Instructor: Dr. Joanne Selinski

What is a program?

- Set of instructions
- Input, processing, output
- Unambiguous
- Terminating
- Ordered

What is a program?

- Set of ordered instructions
- Solve a problem
- Computer executes
- Unambiguous
- Terminating
- Takes input, processes, creates output

Programming Languages

4) Pseudocode - english-like instructions

3) High-Level:

Ada, Ruby, Lisp, MatLab, R

Scripting: Perl, JavaScript

Object Oriented: Python, Java, C++, C#

Structured: FORTRAN, C, COBOL, Pascal, BASIC

2) Low Level: Assembly - symbolic code at the hardware level, Java Bytecode - assembly for the Java Virtual Machine (JVM)

1) Executable, Machine: Binary - 1s & 0s

Programming Languages

4) Lisp, Prolog, functional

3) High-level: Ada

Scripting: Perl, JavaScript

Object-Oriented: C++, JAVA, Python, C#, Smalltalk

Structured: C, COBOL, FORTRAN, Pascal

2) Assembly - shorthand for low level operations -
control CPU, ALU, main memory (registers)

Java Bytecode = assembly for JVM

1) Binary, executable, machine

Program Translation

- From high level to machine level
- Compilers
 - Take program source, translated into a lower form, given back as a new file, execute the new file (run program)
 - C, FORTRAN, C++, Java compiled to bytecode
- Interpreters
 - Translate while running the program
 - BASIC
 - Java bytecode is interpreted when run program



Programming Phases

Programming Phases

1. Define & analyze: know requirements, clarify inputs & outputs, get any formulas, create a sample run

Programming Phases

1. Define & analyze: know requirements, clarify inputs & outputs, get any formulas, create a sample run
2. Design: overall program structure, algorithms in pseudocode

Programming Phases

1. Define & analyze: know requirements, clarify inputs & outputs, get any formulas, create a sample run
2. Design: overall program structure, algorithms in pseudocode
3. Write program code: type, compile, debug, run

Programming Phases

1. Define & analyze: know requirements, clarify inputs & outputs, get any formulas, create a sample run
2. Design: overall program structure, algorithms in pseudocode
3. Write program code: type, compile, debug, run
4. Testing - find errors, go to step 3
- 5.

Programming Phases

1. Define & analyze: know requirements, clarify inputs & outputs, get any formulas, create a sample run
2. Design: overall program structure, algorithms in pseudocode
3. Write program code: type, compile, debug, run
4. Testing - find errors, go to step 3
5. Documentation - for the user
- 6.

Programming Phases

1. Define & analyze: know requirements, clarify inputs & outputs, get any formulas, create a sample run
2. Design: overall program structure, algorithms in pseudocode
3. Write program code: type, compile, debug, run
4. Testing - find errors, go to step 3
5. Documentation - for the user
6. Maintain/upgrade

Algorithms

- Wash your hair
- Calculate average of three numbers
- Find your homework point sum
- Calculate your GPA

Algorithms: average of 3 numbers

- Input three numbers
 - Add the numbers together
 - Divide the total by 3
 - Voila, average
-
- Prompt for 3 numbers
 - Input num1
 - Input num2, add to num1 => sum
 - Input num3, add to sum => sum
 - Divide sum by 3
 - Output "average is", sum

Algorithms: average of 3 numbers

- Ask for 3 numbers
- Read and store in a, b, c
- Calculate the sum of $a+b+c$
- Divide sum by 3, store as avg
- Display "average is: ", avg

Algorithms: homework point sum version 1

- Prompt for number of assignments
- Read number
- Set score to 0
- Repeat number of assignments times:
 - Prompt for assignment grade
 - Read grade
 - Add grade to score
- Output "sum is", score
- If score > 200
 - Output "pass"
- Otherwise
 - Output "fail"

Algorithms: homework point sum version 2

- Set total to 0
- Prompt for homework grade
- Read grade
- Add grade to total
- Display "homework grade", total
- Ask if last grade
- If no,
 - Return to prompt & repeat

3 types of Program Control

- Sequential statement execution (default)
- Decision statements
- Repetition statements (loops)

All general purpose programming languages must have ways to do these 3 things.

GPA Example

- Problem Analysis & Design in file: [code/gpa.txt](#)
- First java version in file: [code/gpa.java](#)

Java Program Structure

- Class file
 - Data (optional)
 - Method definitions
 - Executable statements
 - Variable declarations
 - Method calls
 - Assignment statements
 - Decision statements
 - Loops

Program class files must have special main method.

JAVA Language Elements

- Comments
 - Not executed - documentation only
 - // single line or /* */ block style
- Reserved words
- Symbols
- Identifiers
 - Used for names of: classes, methods, variables
 - Contain: letters, \$, _, digits but not as first character
- Literal values: numbers, characters, strings
- Case sensitive
- White space

JAVA Applications Programming Interface (API)

- Collection of pre-existing Java software
- Organized into useful class files
- Related class files are in common Packages (folders)
- Sometimes need to be explicitly included in order to be used in your program:

```
import java.packageName.className;
```
- `java.lang.*` classes are always implicitly included
- The JAVA [website](#) contains full list & documentation

Primitive Data Types

- **boolean** - stores true (1) or false (0) values
- **byte** - holds integer in one byte = 8 bits
- **char** - holds a single character
- **short, int, long** - hold integers of varying sizes
- **float, double** - hold real numbers (floating point)

Variables have 4 features: name, type, location in memory and value (once assigned)

Integer Representation

- literal values are whole numbers: 8, 23, -102
- base ten values are converted to binary
- left-most bit is for the sign
 - 0 means positive
 - 1 means negative
- each type (`byte`, `short`, `int`, `long`) has different size in memory
- `int` is used most commonly

Character Representation

- literals must be in single quotes: 'A', '4', '+'
- each character is assigned an integer code
- full set is Unicode System
- extension of original ASCII system
- decimal integer codes are converted to binary
- 'A' to 'Z' have consecutive codes (65-90)
- 'a' to 'z' have consecutive codes (97-122)
- '0' to '9' have consecutive codes (48-57)
- Appendix 1 in textbook

Real Number Representation

- **float** literals must have f: 10f, 23.342F, -102.3f
- **double** literals (more common)
 - can be floating: 10.3, -100.23, .023345
 - or scientific notation: 1.03e1, -1.0023e2, .23345e-1
- stored internally in two parts: $123.45 = .12345e3$
 - 12345 is mantissa - converted to binary
 - 3 is exponent - converted to binary
- **double** has twice as many mantissa bits as a float, so you get more precision

Operators have precedence

- () do inside parentheses first
- - negation
- * multiplication, / division, % mod (remainders)
- + addition, - subtraction
- = assignment

Appendix 2 in textbook

Data Type Conversions

- Look at [code/arithmetic.java](#) for examples of type conversions and uses of arithmetic operators
- Implicit conversions from smaller type to larger:
`double d = 23; // integer constant to double variable`
`23.53 * -14 // -14 converted to -14.0 before *`
- Explicit conversions from larger type to smaller:
`int n = (int) 24.645; // will truncate and store 24`
`(int) (-14 / 2.5) // temp result -5.6 converted to -5`
- Explicit conversions from smaller to larger:
`(double) 23 / 4 // to get 23.0 / 4 which is 5.75`

Misc. data items

- escape sequences: `\` `"` `\\` `\n` `\t`
- `final` for named constants
- assignment combinations: `+=`, `-=`, `*=`, `/=`, `%=`
- `++`, `--`

Common Classes

- System
 - print
 - println
 - printf
- Math
- Scanner
- String

Formatting output (printf)

- We use the printf method to specify output formats:
System.out.printf("My account has \$%.2f in it.%n", 200/3.0);
System.out.printf("%5d %3c%n", 23, 'A');
- General form: System.out.printf(formatstring, argumentlist)
 - formatstring is a string literal with format specifiers
 - argumentlist is comma separated list of data to print
 - argumentlist items must match formatspecifiers in order left to right
- format specifiers are placeholders with format info:
%[argument_index\$][flags][width][.precision]conversion
- conversion is a character to format the argument type
 - s - string
 - c - character
 - d - integer (decimal)
 - e - floating scientific notation
 - f - floating real
 - % - to show % sign
 - n - line separator

More printf details

- format specifiers are placeholders with format info:
`%[argument_index$][flags][width][.precision]conversion`
- `[]` components are optional:
 - `argumentindex`: position of argument in list to use: first is 1\$, 4th is 4\$, etc.
 - flags depend on conversion type
 - width is an integer telling the field width
 - precision limits the #digits, depending on conversion type
- See chapter 5 for more details

Math class

- In java.lang (no need to import)
- Contains lots of useful methods
- Call methods directly with class name (no object)
- Most methods return double data values
- Examples:
 - `Math.pow(12, 3.5) =>`
 - `Math.sqrt(243) =>`
 - `Math.ceil(24.234) => 25.0`
 - `Math.floor(52.958) => 52.0`
 - `Math.round(24.345) => 24.0`

Scanner class for input

- `import java.util.*;`
- Create object of class, init w/`System.in` for keyboard input
- Methods for various data types, separated by whitespace (space, tab, newline)
 - `next()` (String, stopping at first whitespace)
 - `nextInt()` (int)
 - `nextDouble()` (double)
 - `next().charAt(0)` (char)
 - `nextLine()` (String, entire line to end including whitespace)
- examples in [code/inputscan.java](#)

String class basics

- in `java.lang` (no need to import)
- create reference (placeholder): `String s;`
- can initialize with a string literal: `s = "this string";`
- character indices start at 0
- use `+` for concatenation: `s = "this " + "string";`
- common useful methods:
 - `charAt(int index)`
 - `indexOf(int ch)`
 - `length()` // returns actual number of characters
 - `substring(startIndex)` (to end)
 - `substring(start, end+1)` (including start and end)

More Strings

- See [code/strings.java](#) for examples
- use `null` to initialize to no String object
- use `""` to initialize to empty String object
- conversions to other types
 - `Integer.parseInt(String)`
 - `Double.parseDouble(String)`
- conversion from primitive to String:
 - `mynum + ""`
- Do simplified [pig-latin](#) example

Decisions, Decisions

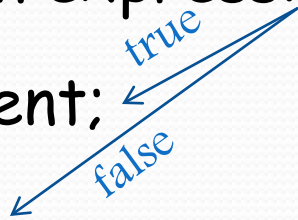
- Decision statements allow us to make a choice based on the result of a test or condition
- Pseudocode example:
 - if age greater than or equal to 16
 - then get driving permit
 - otherwise keep walking
- JAVA has three types of built-in decision statements:
 - if
 - if/else
 - switch
- See [code/decisions.java](#) for examples

One-way if

- General format:

`if` (boolean expression)

statement;



- If the boolean expression is true, the statement gets executed.
- If the boolean expression is false, the statement is skipped.
- Program execution continues with whatever follows.

Two-way if/else

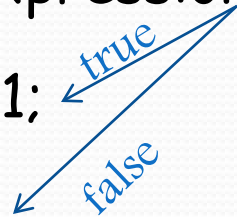
- General format:

if (boolean expression)

statement1;

else

statement2;



- If the boolean expression is true, only statement1 gets executed.
- If the boolean expression is false, only statement2 gets executed.
- Program execution continues with whatever follows.

Block Statements

- In order to make more than one statement be controlled by a decision, we use a block statement
- Curly braces { } are used to define a block
- All statements within the braces are treated as a unit
- A block can be used anywhere a single executable statement is expected
- This permits us to "nest" our statements in any combination

Boolean Expressions

We form boolean expressions with

- comparison operators, for primitive number types
 - < > <= >=
 - == equals
 - != not equals
- logical operators, for combining boolean values
 - ! means not
 - && means and
 - || means or
- boolean methods, when available, for class objects
eg: `word.equals("match")`

Logical Operators

- ! means "not"
!true => false, !false => true
- && means "and"
true && true => true
true && false => false
false && true => false
false && false => false
- || means "or"
true || true => true
true || false => true
false || true => true
false || false => false

DeMorgan's Laws

$$\!(A \ \&\& \ B) \ == \ ! A \ || \ ! B$$

$$\!(\text{raining} \ \&\& \ \text{cold}) \ == \ !\text{raining} \ || \ !\text{cold}$$

$$\!(\text{let} \ >= \ 'A' \ \&\& \ \text{let} \ <= \ 'Z') \ == \ \text{let} \ < \ 'A' \ || \ \text{let} \ > \ 'Z'$$

$$\!(A \ || \ B) \ == \ ! A \ \&\& \ ! B$$

$$\!(\text{Tuesday} \ || \ \text{JAVA}) \ == \ !\text{Tuesday} \ \&\& \ !\text{JAVA}$$

$$\!(\text{ans} \ == \ 'y' \ || \ \text{ans} \ == \ 'Y') \ == \ \text{ans} \ != \ 'y' \ \&\& \ \text{ans} \ != \ 'Y'$$

Operators have precedence

- () do inside parentheses first
- - negation, ! not, (casting), ++, --
- * multiplication, / division, % mod (remainders)
- + addition, - subtraction, + concatenation
- < > <= >=
- == equals != not equal
- && and
- || or
- =, *=, /=, %=, +=, -= assignments

Appendix 2 in textbook

Comparing Strings

Several methods in the String class are helpful:

- `equals(Object obj)` - returns true if contain the exact same chars, false otherwise
- `equalsIgnoreCase(Object obj)` – like equals, ignoring capitalization
- `compareTo(String other)` - returns -, 0, +
- `compareToIgnoreCase(String other)`

- `toLowerCase()` - returns string with all lower case
- `toUpperCase()` - returns string with all upper case

Switch statement

- One way to make multi-way decisions is with a switch
- Sample format (they vary widely):

```
switch (integer expression) {  
  case c1: stmt1; // do stmt1, continue to stmt2  
  case c2: case c3: stmt2; break; // do stmt2, go to end  
  case c4: stmt3; stmt4; break; // do stmt3 & 4, go to end  
  default: stmt5; // do stmt5 if no cases are matched  
}
```
- must control with an integer expression*
- cases (c1, c2, c3, c4) must be integer type constants*
- stmt execution starts with the first matched case and continues through other cases until break or end

*(Java now allows Strings in switches/cases, but we will not be doing this since not all of us have the latest Java compiler installed.)

Testing

- White-box Testing: each possible path in a program (all possible decision cases) should be tested.
- Boundary cases (the = part of \leq or \geq) should be tested.
- Valid values should be tested.
- Invalid values should be tested.
- Regression Testing: when rewriting and updating code, be sure to re-test cases that worked before the "upgrade".

Loops in General

- Loops allow us to repeat one or more statements based on the result of a test or condition
- Loop designs:
 - counter controlled: repeat a certain number of times
 - sentinel controlled: repeat until a value or event occurs
- Pseudocode examples:
 - repeat 10 times (counter controlled)
 - read a number
 - add it to the sum
 - repeat until you run out of input (sentinel controlled)
 - read a number
 - add it to the sum

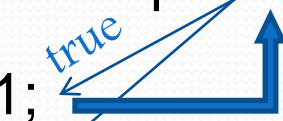

Loops in JAVA

- JAVA has three types of built-in repetition statements:
 - **while**
 - **do/while**
 - **for**
- Each controls one statement
- Use block statement {} to nest and control more than one at a time
- See [code/loops.java](#) for examples

While Loop

- Good for counter or sentinel control
- General form:

while (boolean expression)

statement1; 
statement2; 

- For as long as the boolean expression is true, statement1 gets repeatedly executed
- Whenever the boolean expression is false, the loop ends and the control continues with statement2

Do/While Loop

- Used mostly for sentinel control
- General form:

do

statement1; ← *true*

while (boolean expression);

statement2; ← *false*

- First statement1 is executed, then the boolean expression is tested
- For as long as the expression is true, statement1 will be repeated
- Whenever the boolean expression is false, the loop ends and the control continues with statement2

For Loop

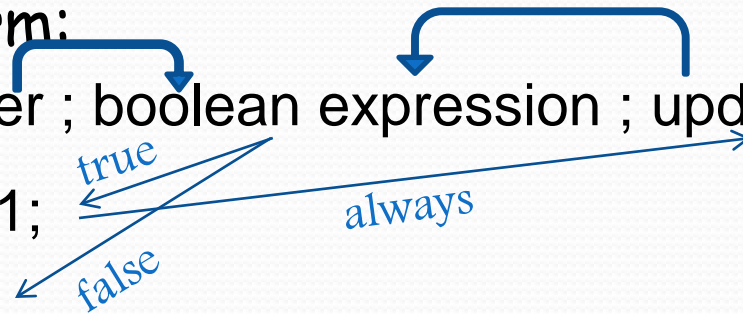
- Used mostly for counter control

- General form:

```
for ( initializer ; boolean expression ; update )
```

```
    statement1;
```

```
    statement2;
```



- First the initializer is executed (only once), then the boolean expression is tested
- For as long as the expression is true, statement1 will be repeated, and after that the update
- Whenever the boolean expression is false, the loop ends and the control continues with statement2

For Loop cont.

- initializer is an expression to initialize the variable(s) controlling the loop
- update is an assignment to change the variable(s) controlling the loop
- initializer list and update can have multiple parts, comma separated
- initializer list and update are optional, but the semicolons are required!

Tutoring Program

- Definition:
 - write a program to help tutor kids in arithmetic
 - give menu with operations: + - * /
 - randomly generate problems
 - give students several chances for each problem:
 - +, - have 1 chance
 - * , / have 3 chances

see <http://www.cs.jhu.edu/~joanne/cs107/code/tutor.java>

Text Files for Input/Output

- text files are simply character streams (sequences)
- standard streams: `System.in`, `System.out`, `System.err` (also for output)
- `import java.io.*;` // I/O package needed for file classes
- use `FileReader` and `FileWriter` classes for character data
- `IOException` must be caught or thrown by every method that indirectly or directly uses an IO class (particularly for input):

```
public static void main(String[] args) throws IOException
```

- writing to a file:

```
PrintWriter outfile = new PrintWriter(new FileWriter("outfile.txt"));  
// proceed as usual: outfile.print(), println(), printf()  
outfile.close(); // very important to save data
```
- use `String` literal or variable with external file name to create
- will overwrite existing contents if file exists, will create new file otherwise
- must close file to save permanently!

Reading from files

- reading from a file:

```
Scanner filescan = new Scanner(new FileReader("infile.txt"));  
// proceed as usual: filescan.next(), nextInt(), nextDouble(), etc.  
infile.close();      // when finished reading from it
```
- use String literal or variable with external file name to initialize
- file must already exist, otherwise an exception is thrown
- reading to end of input:

```
while (scan.hasNext()) // scan can be keyboard or file
```

 - true if there is more input
 - false when the end of input is reached
 - type `^d` (unix) or `^z` (jGRASP) to denote end of keyboard input
- EXAMPLES: [code/filesDemo.java](#)
- There are other types of files that we won't be using in this course.

More Useful JAVA Classes

- Random
- StringBuffer

- BufferedReader - optional material
- StringTokenizer - optional material

Random

- used to generate "random" data
- based on pseudorandom sequence
- in java.util
- must create random object: `Random rand = new Random()`
- uses current time as the seed
- instance methods:
 - `nextInt()` - random integer in the range of (-,+) ints
 - `nextInt(N)` - random int in range [0,N)
 - `nextFloat()` - $0 \leq \text{float} < 1$
 - `nextBoolean()` - returns true or false
 - `nextDouble()` - like `nextFloat()` returning double
- massage data into customized forms with transformations
- examples in [code/randomData.java](#)

Static Methods

- we write static methods to organize programs into sub-tasks
- try to make them independent and re-usable
- method definition = header + body
- method header:
 public static return-type name (parameter list)
- parameter list is comma separated type name pairs
- method body is { statements }
- we use a **return** statement to send a value back to method call
- if a method doesn't return a value, we use **void** as the return type

Method Documentation

- javadoc - program to generate html documentation pages according to a specific format for comments
- put comments immediately preceding class or method definitions
- for a method - list assumptions (preconditions) on params

```
/**
```

General description of method goes on the first line

@param first - first parameter for something

@param second - second parameter for somethingelse

@return - what gets returns if not void, otherwise omit

```
*/
```

- use book icon in jGRASP ("javadoc myprogram.java") to generate myprogram.html file
- view myprogram.html in web browser

StringBuffer

- similar to `String`, but allows character changes, insertions, deletions
- not all `String` methods work directly on `StringBuffer` objects
- initialize with `new StringBuffer(String)`
- `setCharAt(index, value)` - to change a character at a particular index to a new value
- `length()` - the actual number of characters
- `append(*)` - add any primitive, or `String`, to the end of the current value
- see also `insert` & `delete` methods
- `toString()` - to get the simple `String` out of the object

StringBuffer Examples

```
StringBuffer sb = new StringBuffer("hello Student");  
sb.setCharAt(0,'H');  
sb.setCharAt(5,'-');  
System.out.println(sb); // Hello-Student  
// NOT: sb.toUpperCase();  
String s = sb.toString().toUpperCase(); // new String  
// sb not changed
```

- also see [hangman.java](#) code
- remember to check the JAVA API to see what else you can do with this class

BufferedReader

- use for alternative input, instead of Scanner
- in java.io package
- initialize with InputStreamReader for keyboard:

```
BufferedReader br = new BufferedReader(new  
    InputStreamReader(System.in));
```
- initialize with FileReader for text file:

```
BufferedReader br = new BufferedReader(new  
    FileReader(filename));
```
- Two methods for reading:

```
char ch = (char) br.next(); // read 1 char, -1 if end of input  
String line = br.nextLine(); // null if end of input
```

StringTokenizer

- used to break a string into meaningful tokens
- in java.util
- default delimiters are whitespace
- constructors (how to create a new one):
 - StringTokenizer(String arg) (whitespace delims default)
 - StringTokenizer(String arg, String delims) - can specify a different string of delimiters instead of whitespace
- useful methods:
 - countTokens() - number of tokens not yet processed
 - nextToken() - returns substring which is next token
 - hasMoreTokens() - returns true if at least one more

More Practice

- Loops:
 - Tutoring program ([code/tutor.java](#))
 - Shapes program
- Methods & Random
 - html formatting (lab/summer)

Shape Program

- Program to draw a few geometric shapes
- Give user menu of shape choices:
 - triangle
 - square
 - diamond
- Get size and character for each
- Draw it
- [code/shapes.java](#)

Java Class Uses

- Create a program
 - must have main method
 - may have other methods
 - may contain global data (eg: static Random)
- Method library (eg: Math, MyRandom)
 - contains useful static methods
 - may contain [static] data
- Define new data type (eg: String, Card)
 - has instance data
 - has instance methods, including constructor(s)
 - may have class data &/or methods (static)

Method Library Classes

- are not used to create objects
- do not have main methods
- contain useful class methods [**static**]
- may contain some class data [**static**]
- methods are called with the class name
- example: MyRandom.java
- test program: randomData.java

- this type of class is not very common in object oriented programming

Class Definitions

- General form:

```
public class ClassName
{
    // data member (variable) declarations

    // constructor definitions
    public ClassName(param list)
    {
    }

    // method definitions

}
```

- Examples of classes that define new data types
 - [code/Card.java](#), driver program [code/CardMain.java](#)
 - [code/Time.java](#), driver program [code/TimeMain.java](#)

Class Member Types

- Instance
 - data - variables that each individual class object will contain its own copy of and values for
 - method - is called with a class object and may operate directly on the instance data members
- Static (also called "class" members)
 - data - variables that are shared by all objects in the class; there is only one copy
 - method - stand-alone method that does not need any instance members (data or methods) to do its job; called with the class name

Access Specifiers

- **public**
 - can be directly accessed from any other class
 - used for classes, constructors, and many methods
- **protected**
 - can be directly accessed from any other class in the same package (file folder)
 - used for inheritance related classes usually
- **private**
 - can only be directly accessed from within the same class
 - used to protect data
 - used for helper methods

Object class in JAVA

- base of all other classes
- contains three methods:
 - `String toString()`
 - returns class name + memory address
 - `boolean equals(Object o)`
 - compares memory addresses (same as `==`)
 - `Object clone()`
 - returns copy of object
- we override these methods (`toString` & `equals` especially) in our classes to get more appropriate behaviours than the default provided

Arrays

- allow one variable to hold multiple values
- all values must be of the same base type
- integer indices are used to access individual values, called array elements
- [] are used to denote indices
- indices begin at 0 always
- we must declare a size (capacity) to create
- arrays are reference objects in java
- .length tells us the declared size of any array variable

Array examples

- Declare variable:

```
int[] ra;
```

```
double xray[];
```

- Allocate memory:

```
ra = new int[20];
```

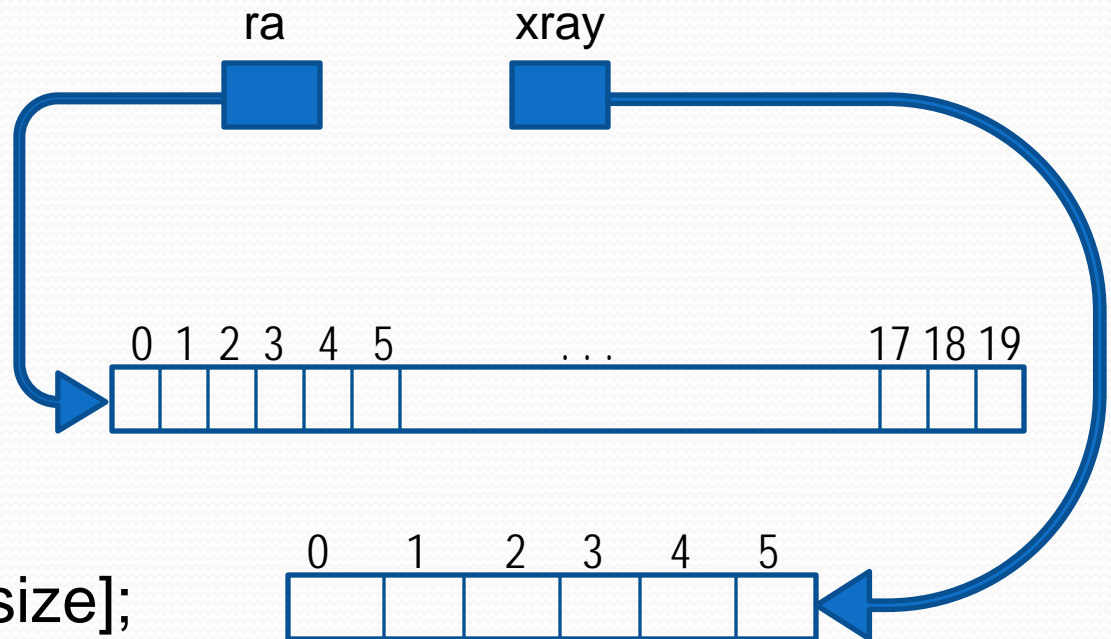
```
int size = 6;
```

```
xray = new double[size];
```

- Access element:

```
ra[0] = 23;
```

```
xray[3] = -23.492;
```



Array Initializations

- primitive base types are initialized to 0
- reference (object) base types are initialized to `null`
- can use an initializer list to declare & initialize if base type is primitive or String:

```
int hw[] = {40, 30, 25, 35, 42, 47};
```

- can use a loop to initialize various ways:

```
for (int i=0; i < hw.length; i++)
```

```
    hw[i] = i*2;
```

2 Dimensional Arrays

- used to hold tabular data
- must have [] for each dimension
 - when referencing individual element
 - when declaring the size
 - first [] refers to the row(s)
 - second [] refers to the column(s)
- is really an array of arrays in Java
- .length is the # of rows
- use nested for loops to manipulate

2D Array Examples

- Declare variable:

```
int[][] ra;
```

```
double xray[][];
```

- Allocate memory:

```
ra = new int[20][5];
```

```
int size = 6;
```

```
xray = new double[size][size*2];
```

- Access element:

```
ra[0][0] = 23;
```

```
xray[3][5] = -23.492;
```

Arrays and Methods

- Can pass whole array to a method
 - parameter type must have matching []
 - eg: `public [static] void print(int[] ra)`
- Array contents are passed by reference
 - changes in the method affect the original contents
- Can return array from a method
 - return type must have matching []

```
public [static] double[][] create(int rows, int cols)
{ double[][] table = new double[rows][cols];
  return table;
}
```
- see [code/arrays.java](#)
- see [code/grades.java](#)

Arrays and Objects

- can make an array of an object type
 - array elements are null references only
 - must say `new BaseType()` to create objects for them
- can put an array inside a class type
 - usually create array memory in constructor
- see [code/Student.java](#), [code/StudentGrades.java](#)
- see [code/Course.java](#), [code/CourseTest.java](#)
- see [code/Deck.java](#), [code/DeckTest.java](#)

WordSearch program

- Define: Create a program to do a word search puzzle from an input file (next slide)
- Design:
 - Objects: word, puzzle, position?
 - Puzzle Methods: read grid from file, get list of words, findWord, searchHorizontal, searchVertical, searchDiagonal, markPosition

WordSearch input file

5 4

javb

abce

vege

atam

twre

Rag

Java

Bee

bet

Common Array Operations

- Initialize (see above)
- Display/print
- Copy
- Resize
- Insert
- Delete
- Search
- Sort

Searching Arrays

- If array is in no particular order:
 - Linear Search
 - go item by item until found, or reach end
- If array is in a particular order (sorted):
 - Binary Search
 - fastest way to win the hi-lo game
 - pick middle value, go left or go right or found

Sorting Algorithms

- Insertion sort: consider each element, move to left as far as it needs to go
- Selection sort: find next largest, swap into position, repeat
- Bubble sort: compare adjacent values, swap if out of order, largest values bubble to the end
- Bucket sort: bins for particular values, sort bins
- Mergesort: split collection in half, mergesort each half, merge them together
- look at [code/Sorter.java](#), [code/SorterMain.java](#)

Algorithm Efficiencies

- Measured as functions of the problem size
- Usually do a worst case analysis
- Space
 - how much (extra) memory is used up?
 - recursive methods can be bad in this respect
- Time
 - primary way we compare algorithms
 - how many basic operations (=, arith, compare, etc.)
 - overall tells us how fast or slow is the algorithm
- Big-Oh: upper bounds on efficiency functions

Sorting Efficiencies

- For array problems, size N elements in array
- Bubble Sort
 - $N-1 + N-2 + \dots + 2 + 1$ ops = $N(N-1) / 2 = O(N^2)$
- Selection Sort
 - $N-1 + N-2 + \dots + 2 + 1$ ops = $N(N-1) / 2 = O(N^2)$
- MergeSort
 - $N * \#$ levels ops
 - $\#$ levels = $\#$ times can divide N by 2 = $\log_2 N$
 - $O(N \log_2 N)$
 - significantly faster than the others

Searching Efficiencies

- For array problems, size N elements in array
- Worst case: value is not there
- Linear Search
 - must compare to all N elements
 - $O(N)$
 - called "linear time", hence "linear search"
- Binary Search (only works if array is sorted)
 - each comparison eliminates $1/2$ the collection
 - # comparisons = # times can divide N by 2
 - $O(\log_2 N)$

Recursion

- solve problem by breaking into smaller pieces
- solve each piece with same strategy
- put pieces together for original solution
- a way of doing repetition

- Recursive method
 - call(s) itself - recursive case(s)
 - base case(s) - does not call itself

- Examples: [recursion.java](#), [balanced.java](#)

Polymorphism

- literally: "many forms "
- key feature of object oriented programming
- method overloading: same name, different parameters (number and/or type)
- Interface
 - defines common set of operations
 - implementations differ from class to class
- Inheritance
 - creates a new [sub, derived] class by extending existing [super, base] one
 - method overriding: redefine method in subclass that exists in superclass

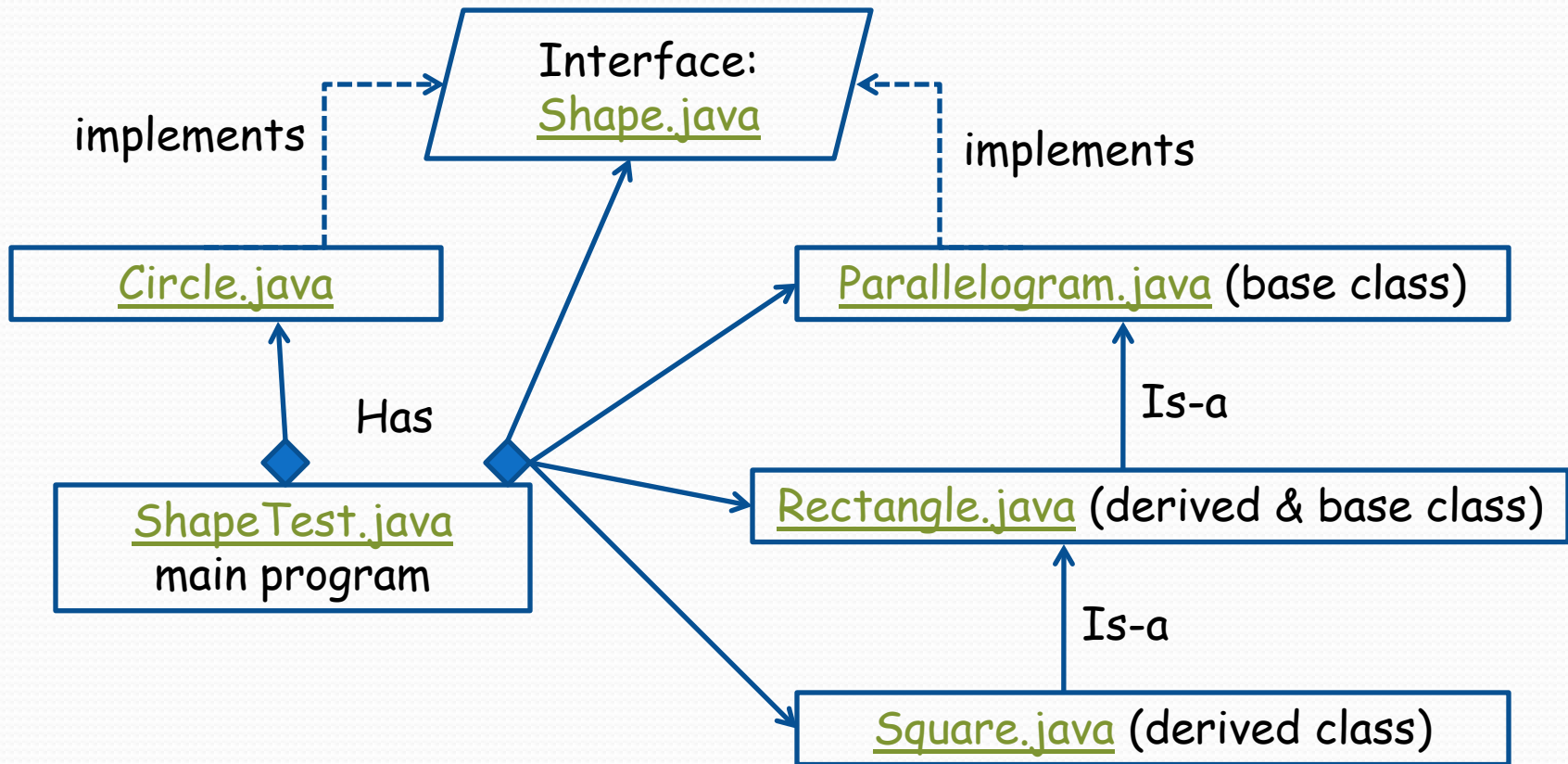
Object class in JAVA

- base of all other classes
- contains three methods:
 - `String toString()`
 - returns class name + memory address
 - `boolean equals(Object o)`
 - compares memory addresses (same as `==`)
 - `Object clone()`
 - returns copy of object
- we override these methods (`toString` & `equals` especially) in our classes to get more appropriate behaviours than the default provided

Polymorphism related reserved words

- **extends**: to say subclass extends base class
- **super**: to call base class constructor or overridden method in base class
- **instanceof**: to see if an object belongs to a particular class
- **protected**: to give access to all other classes in the same package (folder)
- **interface**: to define an interface
- **implements**: to say a class will have method definitions to support an interface

Polymorphism Examples



Testing, revisited

- White-box Testing: each possible path in a program (all possible decision cases) should be tested.
- Black-box Testing: test problem requirements, ignoring code
- Boundary cases (the = part of \leq or \geq) should be tested.
- Valid values should be tested.
- Invalid values should be tested.

Testing, continued

- Regression Testing: when rewriting and updating code, be sure to re-test cases that worked before the "upgrade".
- Unit Testing: individually test each method with it's own mini driver program - incorporate white-box testing
- Testing Data
 - make up literal values
 - random values - helps with black-box testing
 - loops to generate data

Error Handling

- dealing with run-time errors: invalid data, file problems, invalid operations (eg - compare String to a Rectangle)
- test, display error messages: `System.err.print()`
- test, use exception handling
- test, bail out of the program:
 - `(new Throwable()).printStackTrace();`
 - `System.exit(1); // make program stop running`

Exception Handling

- exceptions are objects
- hierarchy of exception classes in Java API
- we can create our own exception classes too
- to use exceptions:
 - create a **try** block - inside there
 - create an exception object (constructor call)
 - **throw** an exception object
 - **catch** an exception object to handle gracefully
 - use **finally** clause to execute code no matter what
- if operation "**throw**"s an exception that is not caught, program will stop and printStackTrace
- to declare exceptions: **throws** particularException
 - needed for checked exceptions (compiler checks)

JAVA Exception Classes

- Throwable (base of exception hierarchy) has methods inherited by other exception classes:
 - Throwable(), Throwable(String msg)
 - getMessage()
 - printStackTrace() , printStackTrace(PrintWriter)
 - toString()
- Exception extends Throwable
- most of these are unchecked
- checked exceptions are often related to IO

More Exception Classes

- unchecked:
 - RuntimeException extends Exception
 - ArithmeticException extends RuntimeException
 - IllegalArgumentException extends RuntimeException
 - NumberFormatException extends IllegalArgumentException
 - [Array/String]OutOfBoundsException
 - NullPointerException
 - InputMismatchException (Scanner)
- checked:
 - IOException
 - FileNotFoundException extends IOException

Exception Examples

- exceptions1.java - has a few JAVA examples
- NotEvenException.java - our own exception class
- exceptions2.java - uses JAVA exceptions and our own NotEvenException
- InvalidCardException.java - Card exception class
- Card.java - edit to throw InvalidCardExceptions