Condensed Java
Python and Java

- Python and Java are both object-oriented languages
  - Conceptually, the languages are very similar
  - The syntax, however, is quite different, and Java syntax is much more complicated
  - Now that you understand the concepts of object-oriented languages, we’ll spend a lot of time on Java syntax

- Java and Python are both popular languages
  - For technical reasons, Java can be much faster than Python
A program, or project, consists of one or more packages
- Package = directory = folder

A package contains one or more classes

A class contains one or more fields and methods
- A method contains declarations and statements

Classes and methods may also contain comments

We’ll begin by looking at the “insides” of methods

Project:
- packages
  - classes
    - fields
    - methods
    - declarations
    - statements
Java structure and Eclipse

- A **workspace** is where Eclipse keeps projects
- When you use Eclipse to create a **project** (a single “program”), it creates a directory with that name in your workspace
- Within the project, you next create a **package**
- Finally, you create a **class** in that package

- For the simplest program, you need only a single package, and only one (or a very few) classes
Simple program outline

```java
class MyClass {
    public static void main(String[] args) {
        new MyClass().run();
    }
    void run() {
        // some declarations and statements go here
        // this is the part we will talk about today
    }
}
```

- **Notes:**
  - The class name (**MyClass**) must begin with a capital
  - **main** and **run** are methods (the name **main** is special; the name **run** isn’t)
  - This is the form we will use for now
    - Once you understand all the parts, you can vary things
What you need to know

- You need to be able to:
  - Read in data (for now, numbers)
    - We will use a Scanner for this
  - Save numbers in variables
    - We will use declarations to create variables
  - Do arithmetic on numbers to get new numbers
    - We will use assignment statements
  - Test whether or not to do something
    - We will use if statements
  - Do something repeatedly
    - We will use while statements
  - Print output
    - We will use System.out.print and System.out.println
  - Use methods
    - A “method” is a function that belongs to a class
Declarations, statements, comments

- A declaration gives type information to the computer
  - You must declare:
    - The type of value (int, String, etc.) each variable can hold
    - The type of every parameter to a method
    - The type returned by every method

- A statement tells the computer to do something
  - Statements should really be called “commands”
  - Statements may only occur within methods

- Comments are ignored by the compiler
  - As in Python, there are different comments for people who use your methods, and those who read your code
Comments

- Python: Single-line comments start with `#`
- Java: Single-line comments start with `//`
- Java: Multi-line comment start with `/*` and end with `*/`
- Python: Documentation comments are enclosed in triple quotes, and are put right after the `def` line
- Java: Documentation comments start with `/**` and end with `*/`, and are put just before the definition of a variable, method, or class
  - Documentation comments are more heavily used in Java, and there are much better tools for working with them
Every variable has a **name**
- Examples: `name`, `age`, `address`, `isMarried`
- Variables start with a lowercase letter
- Multiword variables are written using “camelCase”

Every variable has a **type** of value that it can hold
- For example,
  - `name` might be a variable that holds a `String`
  - `age` might be a variable that holds an `int`
  - `isMarried` might be a variable that holds a `boolean`
    - Boolean constants are `true` and `false` (not `True` and `False`)
- The type of a variable **cannot be changed**
  - However, you might have a different variable with the same name somewhere else in the program
Some Java data types

- In Java, the most important **primitive** (simple) types are:
  - `int` variables hold integer values
  - `double` variables hold floating-point numbers (numbers containing a decimal point)
  - `boolean` variables hold a **true** or **false** value

- Other primitive types are
  - `char` variables hold single characters
  - `float` variables hold less accurate floating-point numbers
  - `byte`, `short` and `long` hold integers with fewer or more digits

- Another important type is the **String**
  - A **String** is an **Object**, not a primitive type
  - A **String** is composed of zero or more **chars**
Declaring variables

- In Python, a variable may hold a value of any type
- Every variable that you use in a program must be declared (in a declaration)
  - The declaration specifies the type of the variable
  - The declaration may give the variable an initial value
- Examples:
  - int age;
  - int count = 0;
  - double distance = 37.95;
  - boolean isReadOnly = true;
  - String greeting = "Welcome to CIT 591";
  - String outputLine;
Multiple values

- An **array** lets you associate one name with a fixed (but possibly large) number of values
- Arrays are like Python’s lists, but much less flexible
- All values must have the same type
- The values are distinguished by a numerical **index** between 0 and array size minus 1

<table>
<thead>
<tr>
<th>myArray</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>12</td>
<td>43</td>
<td>6</td>
<td>83</td>
<td>14</td>
<td>-57</td>
<td>109</td>
<td>12</td>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>
# Using array elements

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>43</td>
<td>6</td>
<td>83</td>
<td>14</td>
<td>-57</td>
<td>109</td>
<td>12</td>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>

- **x = myArray[1];**  
  // sets x to 43
- **myArray[4] = 99;**  
  // replaces 14 with 99
- **m = 5;**  
  **y = myArray[m];**  
  // sets y to -57
- **z = myArray[myArray[9]];**  
  // sets z to 109
To **declare** an array is to tell its type, but *not* its size
- Example: `int[] scores;`

To **define** an array is to give its size
- Example: `scores = new int[40];`

Declaration and definition can be combined
- Example: `int[] scores = new int[40];`

The initial content of an array is zero (for numbers), `false` (for booleans), or `null` (for objects)
Two ways to declare arrays

- You can declare more than one variable in the same declaration:
  ```
  int a[ ], b, c[ ], d; // notice position of brackets
  - a and c are int arrays
  - b and d are just ints
  ```

- Another syntax:
  ```
  int [ ] a, b, c, d; // notice position of brackets
  - a, b, c and d are int arrays
  - When the brackets come before the first variable, they apply to all variables in the list
  ```

- But...
  - In Java, we typically declare each variable separately
Arrays of arrays

- The elements of an array can themselves be arrays
- Once again, there is a special syntax
- Declaration:  `int[][ ] table;` (or `int table[][ ];`)
- Definition: `table = new int[10][15];`
- Combined: `int[][ ] table = new int[10][15];`
- The first index (10) is usually called the row index; the second index (15) is the column index
- An array like this is called a two-dimensional array
Reading in numbers

- First, import the Scanner class:
  ```java
  import java.util.Scanner;
  ```

- Create a scanner and assign it to a variable:
  ```java
  Scanner scanner = new Scanner(System.in);
  ```
  - The name of our scanner is `scanner`
  - `new Scanner(...)` says to make a new one
  - `System.in` says the scanner is to take input from the keyboard

- Next, it’s polite to tell the user what is expected:
  ```java
  System.out.print("Enter a number: ");
  ```

- Finally, read in the number:
  ```java
  myNumber = scanner.nextInt();
  ```
  - If you haven’t previously declared the variable `myNumber`, you can do it when you read in the number:
    ```java
    int myNumber = scanner.nextInt();
    ```
Printing

- There are two methods you can use for printing:
  - `System.out.println(something);`
    - This prints something and ends the line
  - `System.out.print(something);`
    - This prints something and doesn’t end the line (so the next thing you print will go on the same line)

- These methods will print any one thing, but only one at a time
- You can concatenate Strings with the + operator
- Anything concatenated with a String is automatically converted to a String
  - Example:
    ```java
    System.out.println("There are " + appleCount + 
    " apples and " + orangeCount + 
    " oranges.");
    ```
Program to double a number

import java.util.Scanner;

public class Doubler {

    public static void main(String[] args) {
        new Doubler().run();
    }

    private void run() {
        Scanner scanner;
        int number;
        int doubledNumber;

        scanner = new Scanner(System.in);
        System.out.print("Enter a number: ");
        number = scanner.nextInt();
        doubledNumber = 2 * number;
        System.out.println("Twice " + number + " is " + doubledNumber);
    }

}
Assignment statements

- Values can be assigned to variables by assignment statements
  - The syntax is: `variable = expression;`
  - The expression must be of the same type as the variable or a type that can be converted without loss of precision
  - The expression may be a simple value or it may involve computation
  - Examples:
    - `name = "Dave";`
    - `count = count + 1;`
    - `area = (4.0 / 3.0) * 3.1416 * radius * radius;`
    - `isReadOnly = false;`
- When a variable is assigned a value, the old value is discarded and totally forgotten
Organization of a class

- A class may contain data declarations and methods (and constructors, which are like methods), but **not** statements.
- A method may contain (temporary) data declarations and statements.
- A common error:
  - ```
    class Example {
      int variable;            // simple declaration is OK
      int anotherVariable= 5;  // declaration with initialization is OK
      yetAnotherVariable = 5;  // statement! This is a syntax error
    }
    void someMethod( ) {
      int yetAnotherVariable; //declaration is OK
      yetAnotherVariable = 5; // statement inside method is OK
    }
  ```
Arithmetic expressions may contain:

- + to indicate addition
- - to indicate subtraction
- * to indicate multiplication
- / to indicate division
- % to indicate remainder of a division (integers only)
- Sorry, no exponentiation
- parentheses ( ) to indicate the order in which to do things
- Exponentiation is done with the method `Math.pow(base, exponent)`

An operation involving two `int`s results in an `int`
- When dividing one `int` by another, the fractional part of the result is thrown away: `14 / 5` gives 2

Any operation involving a `double` results in a `double`:
- `14.0 / 5` gives approximately 2.8
Boolean expressions

- Arithmetic **comparisons** result in a **boolean** value of **true** or **false**
  - There are six comparison operators:
    - `<` less than
    - `<=` less than or equals
    - `>` greater than
    - `>=` greater than or equals
    - `==` equals
    - `!=` not equals

- There are three **boolean operators**:
  - `&&` “and”--true only if both operands are true
  - `||` “or”--true if either operand is true
  - `!` “not”--reverses the truth value of its one operand

- Example:
  - `(x > 0) && !(x > 99)`
    - “x is greater than zero and is not greater than 99”
You can concatenate (join together) Strings with the + operator

- Example: `fullName = firstName + " " + lastName;`

In fact, you can concatenate any value with a String and that value will automatically be turned into a String

- Example:
  `System.out.println("There are " + count + " apples.");`

Be careful, because + also still means addition

- `int x = 3;`  
  `int y = 5;`  
  `System.out.println(x + y + " != " + x + y);`

  - The above prints `8 != 35`
  - "Addition" is done left to right--use parentheses to change the order
An **if statement** lets you choose whether or not to execute one statement, based on a *boolean* condition

- Syntax:  
  ```java
  if (boolean_condition) {
    statement;
  }
  ```

- Example:  
  ```java
  if (x < 100) {  // adds 1 to x, but only if x is less than 100
    x = x + 1;
  }
  ```

- Python programmers: The parentheses are required

- C programmers:  
  - The condition *must* be boolean  
  - The braces, `{ }`, should be on the lines as shown above. *Do not* insist on using C conventions in Java!
if-else statements

- An if statement may have an optional else part, to be executed if the boolean condition is false
  - Syntax:
    ```java
    if (boolean_condition) {
      statement;
    } else {
      statement;
    }
    ```
  - Example:
    ```java
    if (x >= 0 && x < limit) {
      y = x / limit;
    } else {
      System.out.println("x is out of range: " + x);
    }
    ```
  - Java has no equivalent to Python’s elif (just use else if instead)
Compound statements

- Multiple statements can be grouped into a single statement by surrounding them with braces, \{ \}.

- Example:

  ```java
  if (score > 100) {
    score = 100;
    System.out.println("score has been adjusted");
  }
  ```

- Unlike other statements, there is no semicolon after a compound statement.

- Braces can also be used around a single statement, or no statements at all (to form an “empty” statement).

- Indentation and spacing should be as shown in the above example.
while loops

A while loop will execute the enclosed statement as long as a boolean condition remains true
- Syntax: while (boolean_condition) {
  statement;
}
- Example:
  n = 1;
  while (n < 4) {
    System.out.println(n + " squared is " + (n * n));
    n = n + 1;
  }
- Result:
  1 squared is 1
  2 squared is 4
  3 squared is 9
- Python programmers: The parentheses are required
- C programmers: The condition must be boolean
- Danger: If the condition never becomes false, the loop never exits, and the program never stops
A complete program

```java
public class SquareRoots {
    // Prints the square roots of numbers 1 to 10
    public static void main(String args[]) {
        int n = 1;
        while (n <= 10) {
            System.out.println(n + " " + Math.sqrt(n));
            n = n + 1;
        }
    }
}
```

<table>
<thead>
<tr>
<th>n</th>
<th>Square Root</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>1.4142135623730951</td>
</tr>
<tr>
<td>3</td>
<td>1.7320508075688772</td>
</tr>
<tr>
<td>4</td>
<td>2.0</td>
</tr>
<tr>
<td>5</td>
<td>2.23606797749979</td>
</tr>
</tbody>
</table>

etc.
The **do-while** loop

- The syntax for the **do-while** is:
  ```java
do {
    ...any number of statements...
} while (condition) ;
```
- The **while** loop performs the test first, before executing the statement
- The **do-while** statement performs the test *afterwards*
- As long as the test is **true**, the statements in the loop are executed again
The increment operator

- `++` adds 1 to a variable
  - It can be used as a statement by itself
  - It can be used as part of a larger expression, but this is very bad style (see next slide)
  - It can be put *before* or *after* a variable
    - If before a variable (*preincrement*), it means to add one to the variable, then use the result
    - If put after a variable (*postincrement*), it means to use the current value of the variable, then add one to the variable
    - When used as a statement, preincrement and postincrement have identical results
Examples of ++

```c
int a = 5;
a++;  // a is now 6

int b = 5;
++b;  // b is now 6

int c = 5;
int d = ++c;  // c is 6, d is 6

int e = 5;
int f = e++;  // e is 6, f is 5

int x = 10;
int y = 100;
int z = ++x + y++;  // x is 11, y is 101, z is 111
```

Confusing code is bad code, so this is very poor style
The decrement operator

- -- subtracts 1 from a variable
  - It acts just like ++, and has all the same problems
The **for** loop

- The **for** loop is complicated, but *very* handy
- Syntax:
  ```
  for (initialize ; test ; increment) statement ;
  ```
  - Notice that there is no semicolon after the *increment*
- Execution:
  - The *initialize* part is done first and only once
  - The *test* is performed; as long as it is *true*,
    - The *statement* is executed
    - The *increment* is executed
Parts of the **for** loop

- **Initialize**: In this part you define the loop variable with an assignment statement, or with a declaration and initialization
  - Examples:  \( i = 0 \), \( \text{int} \ i = 0 \), \( i = 0, j = k + 1 \)

- **Test, or condition**: A boolean condition
  - Just like in the other control statements we have used

- **Increment**: An assignment to the loop variable, or an application of \( ++ \) or \( -- \) to the loop variable
  - This may be the *only* good use of \( ++ \) and \( -- \)!

Example for loops

- Print the numbers 1 through 10, and their squares:
  ```java
  for (int i = 1; i < 11; i++) {
      System.out.println(i + "  " + (i * i));
  }
  ```

- Print the squares of the first 100 integers, ten per line:
  ```java
  for (int i = 1; i < 101; i++) {
      System.out.print("  " + (i * i));
      if (i % 10 == 0) System.out.println();
  }
  ```
```java
public static void main(String[] args) {
    for (int i = 1; i < 11; i++) {
        for (int j = 1; j < 11; j++) {
            int product = i * j;
            if (product < 10)
                System.out.print("   "+product);
            else System.out.print("  "+product);
        }
        System.out.println();
    }
    System.out.println();
}
```
When do you use each loop?

- Use the **for** loop if you know ahead of time how many times you want to go through the loop
  - Example: Stepping through an array
  - Example: Print a 12-month calendar

- Use the **while** loop in almost all other cases
  - Example: Compute the next step in an approximation until you get close enough

- Use the **do-while** loop if you must go through the loop at least once before it makes sense to do the test
  - Example: Ask for the password until user gets it right
The break statement

- Inside any loop, the `break` statement will immediately get you out of the loop
  - If you are in nested loops, `break` gets you out of the *innermost* loop

- It doesn’t make any sense to break out of a loop unconditionally—you should do it only as the result of an *if* test

- Example:
  ```java
  for (int i = 1; i <= 12; i++) {
      if (badEgg(i)) break;
  }
  
  `break` is not the normal way to leave a loop
  - Use it when necessary, but don’t overuse it
The continue statement

- Inside any loop, the **continue** statement will start the next pass through the loop
  - In a **while** or **do-while** loop, the **continue** statement will bring you to the test
  - In a **for** loop, the **continue** statement will bring you to the increment, *then* to the test
Multiway decisions

- The **if-else** statement chooses one of two statements, based on the value of a **boolean** expression.

- The **switch** statement chooses one of several statements, based on the value on an integer (**int**, **byte**, **short**, **long**, or **enum**) or a **char** expression.
Syntax of the `switch` statement

- The syntax is:
  ```java
  switch (expression) {
    case value1:
      statements;
      break;
    case value2:
      statements;
      break;
    ...(more cases)...
    default:
      statements;
      break;
  }
  ```

- The `expression` must yield an integer or a character
- Each `value` must be a literal integer or character
- Notice that colons (`:`) are used as well as semicolons
- The last statement in every case should be a `break;
- I even like to do this in the last case
- The `default:` case handles every value not otherwise handled
Flowchart for switch statement

1. Check expression
2. If true, go to corresponding value
3. If false, go to next value
4. Repeat until a match is found
5. Execute corresponding statement
Flowchart for **switch** statement

Oops: If you forget a **break**, one case runs into the next!
Example \texttt{switch} statement

```java
switch (cardValue) {
    case 1:
        System.out.print("Ace");
        break;
    case 11:
        System.out.print("Jack");
        break;
    case 12:
        System.out.print("Queen");
        break;
    case 13:
        System.out.print("King");
        break;
    default:
        System.out.print(cardValue);
        break;
}
```
The **assert** statement

- The purpose of the **assert** statement is to document something you believe to be true
- There are two forms of the **assert** statement:
  1. **assert booleanExpression;**
     - This statement tests the boolean expression
     - It does nothing if the boolean expression evaluates to **true**
     - If the boolean expression evaluates to **false**, this statement throws an `AssertionError`
  2. **assert booleanExpression : expression;**
     - This form acts just like the first form
     - In addition, if the boolean expression evaluates to **false**, the second expression is used as a detail message for the `AssertionError`
     - The second expression may be of any type except **void**
Enabling assertions

- By default, Java has assertions disabled—that is, it ignores them
  - This is for efficiency
  - Once the program is completely debugged and given to the customer, nothing more will go wrong, so you don’t need the assertions any more
    - Yeah, right!
- You can change this default
  - Open Window → Preferences → Java → Installed JREs
  - Select the JRE you are using (probably 1.7.something)
  - Click Edit…
  - For Default VM Arguments, enter -ea (enable assertions)
  - Click OK (twice) to finish
How to define a method

- A method is a “function” that belongs to a class
- The syntax for a method is
  
  ```java
  returnType name(type parameter, ..., type parameter) {
    declarations and statements
    return value;
  }
  ```

- The `returnType` may be `void`, meaning nothing is returned
  - In this case, the `return` statement is unnecessary and, if used, must not provide a value
  - `return` statements may occur anywhere within the method, but the best style is to have only one, at the end

- Types are specified when the method is defined, *not* when it is called

- You may prefix instance variables with `this`, but it’s seldom necessary
A method is a “function” that belongs to a class

The syntax for a method is

```
returnType name(type parameter, ..., type parameter) {
    declarations and statements
    return value;
}
```

You must declare the types of parameters and the return type

- A return type of `void` means nothing is returned
- If a method returns `void` (nothing), you may use `return;` statements in it
- If you reach the end a `void` method, it automatically returns

Java’s keyword “this” is the same as Python’s variable “self”

- In Java, never write `this` (or `self`) as the first parameter—it’s always available
How to “call” a method

- A **method call** is a request to an object to do something, or to compute a value
  - `System.out.print(expression)` is a method call; you are asking the `System.out` object to evaluate and display the `expression`

- When you call a method, do **not** specify parameter types
  - You must provide parameters of the type specified in the method definition

- Any method call may be used as a statement
  - Example: `System.out.print(2 * pi * radius);`
  - If the method returns a value, the value is ignored

- Methods that return a value may be used as part of an expression
  - Example: `h = Math.sqrt(a * a + b * b);`
  - To use the correct technical jargon, we don’t “call” methods; we **send a message** to the object that holds the method
import java.util.Random;

public class RandomWalk {
    int x = 0;
    int y = 0;
    Random rand = new Random();

    public static void main(String[] args) {
        new RandomWalk().run();
    }

    void run() {
        double distance = 0;
        while (distance < 10) {
            step(3);
            System.out.println("Now at " + x + ", " + y);
            distance = getDistance();
        }
    }

    void step(int maxStep) {
        x += centerAtZero(maxStep);
        y += centerAtZero(maxStep);
    }

    int centerAtZero(int maxStep) {
        int r = rand.nextInt(2 * maxStep + 1);
        return r - maxStep;
    }

    double getDistance() {
        return Math.sqrt(x * x + y * y);
    }
}
“I think there is a world market for maybe five computers.”

—Thomas Watson, Chairman of IBM, 1943

“There is no reason anyone would want a computer in their home.”

—Ken Olsen, president/founder of Digital Equipment Corporation, 1977