CSE 1325: Object-Oriented Programming

Lecture 01

From C to Java

Into the OOP Rabbit Hole

(These slides are at Canvas > Modules > Lecture 01)

Mr. George F. Rice

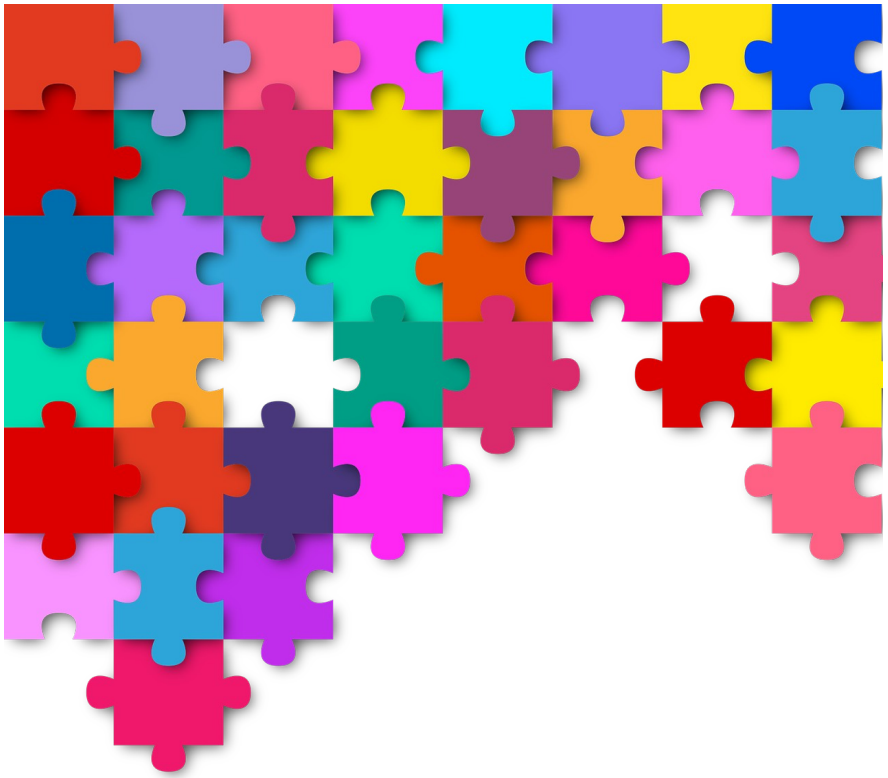
george.rice@uta.edu

Office Hours:
Prof Rice 12:30 Tuesday and Thursday in ERB 336
For TAs see this web page

Texas Linux: Y'all Reckon? (Yep/Nope)

Topic Overview

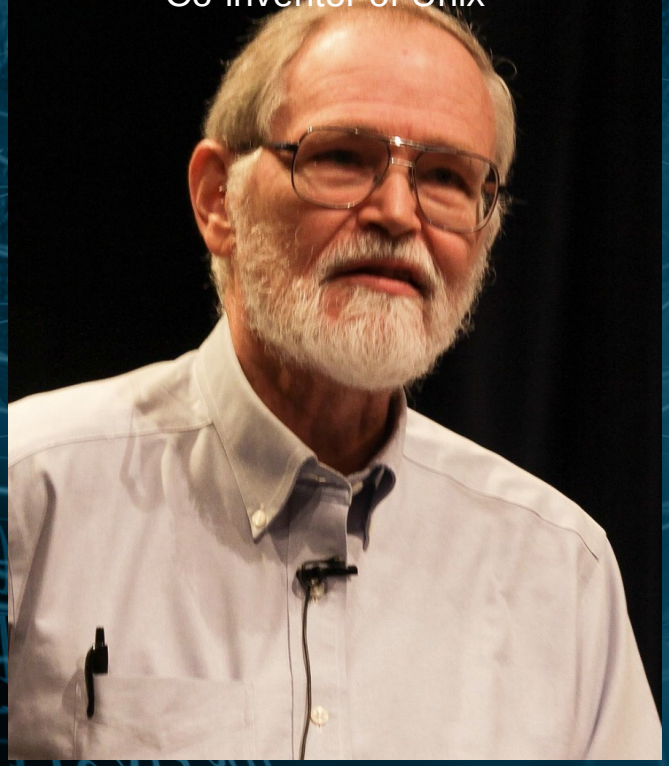
- Java Basics
  - Type Comparison
  - Console I/O
    - println & printf
    - Scanner
    - Console
  - Syntax
  - Parameter Handling
- Packages and Import
- Ant (Automated Builds)
- A bit more git



Java is Similar to C

Java is *Also* a K&R Language!

Brian Kernighan
Professor of CSE at Princeton
Co-inventor of Unix



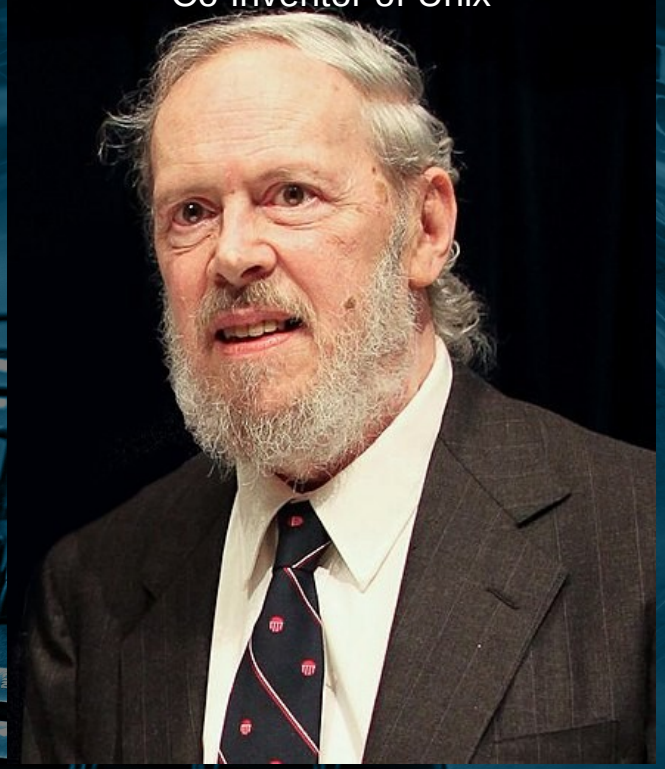
All K&R languages derive from C:

- Start at main()
- { } defines scope
- ; terminates a line
- Loops with 3-term for
  `for(int i=0;i<10;++i){}`

Other K&R include

- C++, C#, Objective C
- JavaScript, Rust
- Go, Perl, R, MATLAB

Dennis Ritchie
Inventor of C
Co-inventor of Unix



Popular *non*-K&R languages include

- Python, Ruby, bash, Visual BASIC, F#, SQL, HTML / CSS

C/C++, Java, and Python Types

Primitives

Type	C / C++	Java	Python
1-byte integer	**char**	byte	**int** All integers are of arbitrary size
2-byte integer	short, **int** (often 4 bytes)	**char**, short	
4-byte integer	long (often int)	**int**, Integer	
8-byte integer	long long	long	
4-byte double	float	float	
8-byte double	**double**	**double**	**float**
8-byte complex			complex
1-byte character	char		bytes
2-byte character	w_char	char	
Boolean	**bool**	**boolean**	**bool**
String	**std::string** char*	**String**	**str**

Hello, World in Java

- To send output to the console, use the print and println *methods* (“functions that manipulate data in a class”)

Hello.java

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

  - `public class Hello` is the *class* name (more on this soon!) and *file* name
  - `public static void main` is equivalent to `int main` in C
    - Unlike in C, the array of String is *required* in Java, and args[0] is the first argument rather than the program name
    - Unlike in C, main does NOT return an int – it is always void
  - `System.out.println` is a *method* (a function inside a *class*) that prints its one parameter to the console
    - The parameter may be any compatible type, most often String
    - `println` appends a newline, `print` does not

Build and Run "Hello, Java"

- The Java compiler *strictly* enforces filenames
  - Class Hello *must* be in file Hello.java to compile*

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

Hello.java

  - The executable is Hello.class, *not* a.out, and is run with the bash command **`java Hello`**

```
ricegf@pluto:~/dev/202008/02/java$ ls
Concat.java  Constants.java  Equals.java  Hello.java  Overflow.java
ricegf@pluto:~/dev/202008/02/java$ javac Hello.java
ricegf@pluto:~/dev/202008/02/java$ ls
Concat.java  Constants.java  Equals.java  Hello.class  Hello.java  Overflow.java
ricegf@pluto:~/dev/202008/02/java$ java Hello
Hello, Java!
ricegf@pluto:~/dev/202008/02/java$
```

* This is true only for *public* classes, but a good rule for *all* general purpose classes regardless of visibility

print and println for Every Type!

- What's a "compatible type"???
  - Any primitive type (int, double, boolean, ...)
  - A String ← The most common choice by far!
  - A custom type that can convert itself to a String
    - We'll cover how a bit later

void	`println()`	Terminates the current line by writing the line separator string.
void	`println(boolean x)`	Prints a boolean and then terminate the line.
void	`println(char x)`	Prints a character and then terminate the line.
void	`println(char[] x)`	Prints an array of characters and then terminate the line.
void	`println(double x)`	Prints a double and then terminate the line.
void	`println(float x)`	Prints a float and then terminate the line.
void	`println(int x)`	Prints an integer and then terminate the line.
void	`println(long x)`	Prints a long and then terminate the line.
void	`println(Object x)`	Prints an Object and then terminate the line.
void	`println(String x)`	Prints a String and then terminate the line.

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/io/PrintStream.html

Printing Multiple Values

- The '+' operator concatenates Strings
- All types convert to a String when used in a "string context" (as if they were a String)

ComplexPrint.java

```
class ComplexPrint {
    public static void main(String[] args) {
        String s = "Hello";
        int i = 42;
        double pi = 3.14159265;
        char c = 'ツ';
        System.out.println(s + ' ' + i + ' ' + pi + ' ' + c);
    }
}
```

```
ricegf@antares:~/dev/202101/temp$ javac ComplexPrint.java
ricegf@antares:~/dev/202101/temp$ java ComplexPrint
Hello 42 3.14159265 ツ
ricegf@antares:~/dev/202101/temp$
```

- Combined, these allow efficient printing in Java

Formatted Output

- Our job is to meet our users' expectations
  - Even if it makes our job harder!
- People are *very* fussy and particular
- Some are downright *picky* about their output formats
  - They often have good reasons to be
  - Convention and tradition = domain-specific vocabularies
    - What does 110 mean?
    - What does 123,456 mean?
    - What does (123) mean?

The world of output formats is weirder than you could possibly imagine

Formatted Output: Printf

- System.out also offers C-inspired printf methods

```java
public class Printf {
    public static void main(String args[])  {
        // Different integer bases
        int i = 1234;
        System.out.printf("Int    as dec %d,        hex %x,           and oct %o\n",
                          i, i, i);

        // Different double bases
        double d = 1234.56789;
        System.out.printf("Double as dec %.6f, hex %.6a, and exp %.6e\n", d, d, d);

        // align right and include 20 characters
        System.out.printf("Right-align with 4 decimal places: | %20.4f|\n", d);
    }
}
```

```
ricegf@pluto:~/dev/202008/02/java$ javac Printf.java
ricegf@pluto:~/dev/202008/02/java$ java Printf
Int    as dec 1234,        hex 4d2,           and oct 2322
Double as dec 1234.567890, hex 0x1.34a458p10, and exp 1.234568e+03
Right-align with 4 decimal places: |            1234.5679|
ricegf@pluto:~/dev/202008/02/java$
```

Note: Java's "sprintf" equivalent is `String.format("%d\n", i);`

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Formatter.html

Reading the Console in Java

- To read data from the console, use a Scanner

JavaInput.java

```
import java.util.Scanner;

public class JavaInput {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.print("Enter two integers: ");
        int num1, num2;
        num1 = in.nextInt();
        num2 = in.nextInt();
        System.out.println("The sum        is " + (num1 + num2));
        System.out.println("The difference is " + (num1 - num2));
        System.out.println("The product    is " + (num1 * num2));
    }
}
```

Got it? OK, we'll stop reminding you about filenames!

- Yeah – that's a lotta code!
  - Next slide – quick!

Breaking Down JavaInput

- `import java.util.Scanner;` is a *little* like `#include <scanner.h>` in C – it lets us use a `Scanner` in our code instead of `java.util.Scanner`
- `Scanner in = new Scanner(System.in);` creates variable `in` holding a Scanner
  - The parameter sets it to use System.in (that is, the keyboard)
- `num1 = in.nextInt();` tells the scanner to read the next `int` from the keyboard and put it into variable `num1`
  - As with println, we have next methods for all of the primitive types

String	next()	Finds and returns the next complete token from this scanner.
String	next(String pattern)	Returns the next token if it matches the pattern constructed from the specified string.
String	next(Pattern pattern)	Returns the next token if it matches the specified pattern.
BigDecimal	nextBigDecimal()	Scans the next token of the input as a BigDecimal.
BigInteger	nextBigInteger()	Scans the next token of the input as a BigInteger.
BigInteger	nextBigInteger(int radix)	Scans the next token of the input as a BigInteger.
boolean	nextBoolean()	Scans the next token of the input into a boolean value and returns that value.
byte	nextByte()	Scans the next token of the input as a byte.
byte	nextByte(int radix)	Scans the next token of the input as a byte.
double	nextDouble()	Scans the next token of the input as a double.
float		
int	nextInt()	Scans the next token of the input as an int.

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Scanner.html

Reading the Console in Java

- This gets us our int values

```java
import java.util.Scanner;

public class JavaInput {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.print("Enter two integers: ");
        int num1, num2;
        num1 = in.nextInt();
        num2 = in.nextInt();
        System.out.println("The sum        is " + (num1 + num2));
        System.out.println("The difference is " + (num1 - num2));
        System.out.println("The product    is " + (num1 * num2));
    }
}
```

```
ricegf@pluto:~/dev/202008/02/java$ javac JavaInput.java
ricegf@pluto:~/dev/202008/02/java$ java JavaInput
Enter two integers: 42 18
The sum        is 60
The difference is 24
The product    is 756
ricegf@pluto:~/dev/202008/02/java$
```

What About newline?

- Scanner also has a version of next that returns an entire newline-terminated input string

```
ricegf@pluto:~/dev/202008/02/java$ javac WhoDat.java
ricegf@pluto:~/dev/202008/02/java$ java WhoDat
Enter your name: George F Rice
Your name is George F Rice
ricegf@pluto:~/dev/202008/02/java$
```

```java
import java.util.Scanner;

public class WhoDat {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        String s1;
        System.out.print("Enter your name: ");
        s1 = in.nextLine();
        System.out.println("Your name is " + s1);
    }
}
```

nextLine() sounds a little like C's
`getline(&buf, &size, stdin);`
Doesn't it?

Note that **next()** reads a whitespace-separated *word* while **nextLine()** reads an entire newline-terminated *line*.
Note that nextLine() consumes the \n, while next() does not.
So be careful when mixing next() and nextLine()!

Reading to End of File

- For Scanner, nextInt relies on hasNextInt to determine data availability – same with Double, Char, and other primitives as well as Line

```java
import java.util.Scanner;

public class GradeCalc {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        double sumOfGrades = 0;
        int numGrades = 0;
        System.out.println("Enter grades. Press Control-d (Linux, Mac)"
            + " or Control-z (Windows) when done.");
        while(in.hasNextDouble()) {
            sumOfGrades += in.nextDouble();
            ++numGrades;
        }
        System.out.println("The student's average grade is "
                           + sumOfGrades / numGrades);
    }
}
```

```
ricegf@antares:~/dev/cse1325-prof/01/code_from_slides$ java GradeCalc
Enter grades. Press Control-d (Linux, Mac) or Control-z (Windows) when done.
88.5 91.0 90.5
The student's average grade is 90.0
ricegf@antares:~/dev/cse1325-prof/01/code_from_slides$
```

Alternative: Simple Console I/O

- Java also includes a Console class that provides a **`printf(String format, Object…)`** as well as **`readLine()`** and **`readLine(String prompt)`** functions (“static methods”).
- This is great for quick I/O, but does NOT include hasNextInt() or nextInt() (although **`Integer.parseInt(console.readLine())`** is roughly equivalent)

```
public class ConsoleIO {
    public static void main(String[] args) {
        java.io.Console console = System.console();      // Get the console, OR
        int a = 42;                                      // var = System.console();
        console.printf("As easy as 1, 2, %d\n", a);
        String s = console.readLine("Don't you think? "); // Prompt and read String
        console.printf("You really think %s???\n", s);    // Standard printf
    }
}
```

```
ricegf@antares:~/dev/202108/02/code_from_slides$ javac ConsoleIO.java
ricegf@antares:~/dev/202108/02/code_from_slides$ java ConsoleIO
As easy as 1, 2, 42
Don't you think? not really
You really think not really???
ricegf@antares:~/dev/202108/02/code_from_slides$
```

Common Java Operators and Relationals

- Java uses standard notation for operators
  - `+ - * /` (int, double) addition, subtraction, multiplication, and division
  - `+` (String) concatenates (- * and / are errors)
  - `+= -= *= /=` (int, double) performs the operator on the target
  - `+=` (String) concatenates the string to the end of the target
- Java uses NON-standard notation for non-primitive comparisons
  - `==` means "the *address* is the same", while
    `.equals` means "the *data* is the same" ← most common

```java
public class Equals {
  public static void main(String[] args) {
    Integer x=32000;
    Integer y=32000;
    if(x == y) System.out.println("x == y");
    if(x.equals(y)) System.out.println("x equals y");
  }
}
```

```
ricegf@pluto:~/dev/202008/02/java$ javac Equals.java
ricegf@pluto:~/dev/202008/02/java$ java Equals
x equals y
ricegf@pluto:~/dev/202008/02/java$
```

http://tutorials.jenkov.com/java/math-operators-and-math-class.html

Names in Java

- Same as C, except
  - $ is also permitted, although discouraged
  - Camel case is preferred in Java
- Examples
  - Java variable: numberOfElements
    C: number_of_elements
  - Java custom type: PlanesTrainsAndAutomobiles
    C: Planes_trains_and_automobiles

Naming Conventions

- Java lacks a standard naming convention. Every project has their own.
- Here's what I typically use with Java (red indicates a change from C):
  - Names are camel case, e.g., no underscores, capitalize each word (myName)
  - Objects and methods (functions) are NOT capitalized (graders.sort(), i, grade)
  - Bools are yes / no questions (database.isRunning())
  - Classes and other types I define are capitalized (CoordinateSystem)
  - Constants and preprocessor variables are shouted (NUM_WORDS)
  - Packages* are lowercase reverse URLs (edu.uta.cse1325.constants)
  - Opening brace is in-line with the scope identifier (`while(true) {`).
    Closing brace is in-line for a single line block (`if (a == 5) {return 42;}`)
    and on its own line otherwise. I sometimes code “bare” (`if (a == 5) return 42;`).
- Other options
  - Google's preferences https://google.github.io/styleguide/javaguide.html
  - Oracle’s (deprecated) conventions
    https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html

* We’ll get to packages in a moment!

Primitive Types in Java

Declaring and Initializing Variables

```java
public class Constants {
  public static void main(String[] args) {
    boolean isHappy = true;
    char teach = '教';
    int ultimateAnswer = 42;
    double earthMass = 5.972e+27; // in grams
    String goodbye = "So long, folks!";
    System.out.println(goodbye + ' ' + isHappy + ' '
        + teach + ' ' + ultimateAnswer +  ' ' + earthMass);
  }
}
```

The String type is NOT a primitive type and char* does NOT exist in Java! String is a custom, NON-primitive type.

```
ricegf@pluto:~/dev/202008/02/java@ javac Constants.java
ricegf@pluto:~/dev/202008/02/java$ java Constants
So long, folks! true 教 42 5.972E27
ricegf@pluto:~/dev/202008/02/java$
```

These are the primitive types (along with NON-primitive String) that you will most often use by far!

Type Safety

- Java compilers don't detect most overflows

```
class Overflow {
  public static void main(String[] args) {
    int x = Integer.MAX_VALUE-3;
    for(int i=0; i<6; ++i)
        System.out.println(++x);
  }
}
```

Note: Integer is a more complex form of an int, with many helpful utilities and constants.

```
ricegf@pluto:~/dev/202008/02/java$ javac Overflow.java
ricegf@pluto:~/dev/202008/02/java$ java Overflow
2147483645
2147483646
2147483647
-2147483648
-2147483647
-2147483646
ricegf@pluto:~/dev/202008/02/java$
```

Types assign *some* semantic information to a variable – you must provide the rest!

Expressions

- An expression is “a sequence of operators and operands that specifies a computation”
  - Mathematical rules of precedence apply: 3+5*4 is 23
  - Parentheses are better! 3+(5*4) is more clearly 23
  - Break complex expressions onto separate lines to simplify debugging – you can see intermediate results readily
- Choose meaningful variable and method names
  - accountBalance is better than ab
  - i and j are counters for historical reasons (why?)

```
// compute area:
int length = 20;      // the simplest expression: a literal (here, 20)
                     // (here used to initialize a variable)
int width = 40;
int area = length*width;  // a multiplication
int average = (length+width)/2;   // addition and division
```

Statements

- A statement is
  - an expression terminated with a semicolon, or
  - a declaration, or
  - a “control statement” that determines the flow of control
- A compound statement encloses zero or more statements in curly braces
  - May be used anywhere a statement is expected
  - Often called "blocks"

- Statements
  - a = b;
  - double d2 = 2.5;
  - if (x == 2) y = 4;
  - int average = (length+width)/2;
  - return x;
- Compound statements
  - {i = 10; while(--i)
    { System.out.println(i);
    }}
  - {operating = true; light = red;}

Selection (if/else if/else), ?

- If selects between alternatives

```
if (a<b)      // Note:  No semicolon here
    sign = 1;
else if (a>b) // Note:  No semicolon here
    sign = -1;
else
    Sign = 0;
```

Note: Coding single statements on if or while without { } is called “bare coding”.
Some frown on this, although I’m rather old school and don’t.
Follow your code guidelines.
Or your conscience.

The Ternary Operator

- The Java ternary (?) operator is an *in-line* if / else expression to select *data* rather than *statements*
  - `String s = (name == “Rice”) ? “Prof” : “Student”;`

    conditional — If true — If false
  - Same as

```
String s;
if (name == “Rice”) s = “Prof”;
else s = “Student”;
```

- Ternaries can be nested
  -

```
S = (name == “Rice”) ?
    ((class == “CSE1325”) ? “Prof” : “Advisor”)
    : “Student”;
```

- These should only be used to *simply* select data within an expression
  - When in doubt, use if / else!

http://tutorials.jenkov.com/java/ternary-operator.html

Selection (switch)

- Switch is a less flexible but arguably more readable if / else construct

```
enum Color {GREEN, YELLOW, RED};
Color color = Color.RED; String c;

switch(color) {
    case Color.RED:    c = "red"; break;
    case Color.YELLOW: c = "yellow"; break;
    case Color.GREEN:  c = "green"; break;
    default:           c = "blinking red";
}
```

Don't forget the **break**!

- Is equivalent to

```
enum Color {GREEN, YELLOW, RED};
Color color = Color.RED; String c;

if      (color == Color.RED)    c = "red";
else if (color == Color.YELLOW) c = "yellow";
else if (color == Color.GREEN)  c = "green";
else                            c = "blinking red";
```

I've indented the conditionals for parallelism here, but I do NOT usually code or recommend *you* code like this.

Selection (switch without break)

- Java 14+ (but <u>not</u> earlier versions or C / C++) can also accept
  - multiple comma-separated terms per case
  - { } instead of breaks
  - using -> instead of :

```
class Switch {
    public float expectedWorkingTime(DayOfWeek dow) {
        // A switch statement with the new ->
        switch (dow) {
            case MONDAY, TUESDAY, WEDNESDAY, THURSDAY -> {return 8f;}
            case FRIDAY -> {return 6f;}
            default -> {return 0f;}
        }
    }
    public static void main(String[] args) {
        Switch sw = new Switch();
        System.out.println(sw.expectedWorkingTime(DayOfWeek.FRIDAY));
    }
}
```

Code adapted from https://www.mscharhag.com/java/jdk14-switch

Selection (switch expressions)

- Java 14+ can also use the new switch as an *expression* (like a ternary) rather than a *statement*
  - Every possible case must be covered (think *default*)

```
class Switch {
    public float expectedWorkingTimeExp(DayOfWeek dow) {
        // A single return statement with a switch expression
        return switch(dow) {
            case MONDAY, TUESDAY, WEDNESDAY, THURSDAY -> 8f;
            case FRIDAY -> 6f;
            default -> 0f;
        };
    }
    public static void main(String[] args) {
        Switch sw = new Switch();
        System.out.println(sw.expectedWorkingTimeExp(DayOfWeek.FRIDAY));
    }
}
```

Selection (switch expressions)

- Switching from switch to switch expression

```
enum Color {GREEN, YELLOW, RED};
Color color = Color.RED;
String c;
switch(color) {
    case Color.RED:    c = "red"; break;
    case Color.YELLOW: c = "yellow"; break;
    case Color.GREEN:  c = "green"; break;
    default:           c = "blinking red";
}
```

Classic Switch

```
enum Color {GREEN, YELLOW, RED};
Color color = Color.RED;

String c = switch(color) {
    case Color.RED    -> "red";
    case Color.YELLOW -> "yellow";
    case Color.GREEN  -> "green";
    default           -> "blinking red";
};
```

Modern Switch Expression

Additional discussion on Java 14 switch *expressions* is available in this blog post:
https://medium.com/better-programming/a-look-at-the-new-switch-expressions-in-java-14-ed209c802ba0

Selection (switch patterns)

- Java 21 can also switch on a variable's *type*, auto-casting it and evaluating a conditional:

```
enum Color{RED, GREEN, BLUE}; // Just like in C

public class Test {
    public static String toString(Object o) { // See Note
        return switch(o) {
            case null -> "null";
            case String s when s.length() == 0 -> "Empty String";
            case String s -> "String: " + s;
            case Color c -> "Color: " + c;
            default -> "Other: " + o.toString();
        };
    }
    public static void main(String[] args) {
        Object o = Color.RED;
        String s = toString(o);
        System.out.println(s);
        System.out.println(toString(""));
    }
}
```

```
@prof-rice → /workspaces/202408 (main) $ java Test
Color: RED
Empty String
@prof-rice → /workspaces/202408 (main) $
```

Note: For now, consider Object to mean "any type that's not primitive"

Iteration (while loop)

- The primary Java loop construct is while

```
// Conditional before the loop
int i = 0;
while (++i<10)  {
    Sys.println(i);
}
```

```
// Conditional after the loop
int i = 0;
do {
    Sys.println(i);
} while (i++ < 10);
```

```
// Conditional mid-loop
int i;
while (true)  {
    i = in.nextInt(); // in is a Scanner
    if (i == 0) break;
    Sys.println(i);
}
```

Some Java professionals disapprove of mid-loop conditionals.

I approve of whatever offers *best readability*.

Follow your coding guidelines.

Iteration (3-term for loop)

- The 3-term for collects all the control information in one place, at the top, where it's easy to see

```
for (int i = 0; i < 100; ++i) {
    System.out.println(i);
}
```

Commonly (but NOT always):

for (initialize**;** condition **;** increment **)**

controlled statement

```
import java.util.Scanner;

public class For3 {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        for(String line = ""; in.hasNextLine() ; ) {
            line = in.nextLine();
            System.out.println(line);
        }
    }
}
```

Unorthodox, but it works!

```
ricegf@antares:~/dev/202308/
> javac For3.java
ricegf@antares:~/dev/202308/
> java For3
Hello
Hello
World
World
How
How
Are
Are
You
You
Type Control-d to exit
Type Control-d to exit
ricegf@antares:~/dev/202308/
```

Iteration (for-each loop)

- The for-each directly iterates over each element of an array (or Collection – more on Collections later)

```
int[] ints = new int[]{15,42,19}; // Java's array syntax
for(int i : ints) System.out.println(i);
```

```
ricegf@antares:~/dev/202308/
> javac ForEach.java
ricegf@antares:~/dev/202308/
> java ForEach
15
42
19
ricegf@antares:~/dev/202308/
```

Which do you find easier to read?

```
int[] ints = new int[]{15,42,19}; // Java's array syntax
for(int i=0; i<ints.length; ++i) System.out.println(ints[i]);
```

A Word of Warning: Packages

- Packages are used to organize code in Java
  - The entire package can be “imported” elsewhere

```
package myPackage;

public class MyClass {
    public static void main(String[] args) {
        System.out.println("Running myPackage.MyClass.main()");
    }
}
```

The code in this file will be in “myPackage”

- **DON’T create packages until we cover them**
  - Packages will break your homework assignments (for now)
  - If your editor adds a package statement, delete it (and fix your editor or change editors)

Importing (Existing) Packages

- To use a package, you always have a choice
  - *Either* use the fully qualified name

```
java.util.Scanner in = new java.util.Scanner(System.in);
```

  - *Or* use import and the bare name (the usual approach)

```
import java.util.Scanner;
Scanner in = new Scanner(System.in);
```

  - These two statements are equivalent
- Because Java forces a public class into a matching filename, and packages always exactly mirror directory paths, javac can efficiently find the file with the referenced type!
- We usually `import` to keep our code concise

Importing (Existing) Packages

- You may reference a class using its full package name

```
public class UseScanner {
    public static void main(String[] args) {
        java.util.Scanner in = new java.util.Scanner(System.in);
        int i = in.nextInt();
    }
}
```

- OR you may import the class and use its “bare” name

```
import java.util.Scanner;

public class MyPackageRunnerImport {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int i = in.nextInt();
    }
}
```

Never `import *`

- You *could* import all members of a package using *
  - `import java.io.*; // Console and all other classes from`
    `// package java.io now in local scope`
  - **But avoid * in imports** – it causes “namespace pollution”
  - It’s common in Java to have a lot of import statements!
- The Java compiler *automatically* imports 2 key packages
  - `java.lang.*`
  - All members of the *current* package

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/package-summary.html

Automating Compiles

- How do you remember all this for every project?
  - **Always use a build tool!** But which?
- C relies on **`make`**, but make is less suitable for Java
  - C compilers start fast, so **`make`** invokes them for every file
    - Java compilers start slowly
    - We want to compile as many files as possible per invocation
  - C projects are typically in fairly flat directory structures
    - **`make`** handles directories somewhat awkwardly as a result
    - Java projects typically have deep directory structures
    - We want to easily handle complex directory structures during builds
- Thus, Another New Tool (ANT) is the Java standard

Why Learn Ant?

- Ant (like **`make`**) is ubiquitous and cross-platform
- Ant (like **`make`**) is flexible enough to automate other routine project management tasks
- Ant (like **`make`**) works well from the command line but is also well-supported by every major Java IDE
- Ant relies on the eXtended Markup Language (XML), a common HTML-like industry standard
- Ant is the basis for more powerful (and complex) build tools such as Maven and Gradle
  - You will learn (a little) Ant quickly and then focus on Java for now
  - Learning Ant will help you learn those tools later

Hello, World in Ant

- The following is “Hello, World!” in Ant

```
<?xml version="1.0"?>
   <project name="Hello World Project" default="info">
   <target name="info">
      <echo>Hello, World!</echo>
   </target>
</project>
```

- `<?xml version="1.0"?>`
  is the *required* “magic cookie” that identifies this as XML
- `<project name="Hello World Project" default="info">`
  and its closing tag `</target>` specify the name of the project and the default “target” to be built if we simply execute “ant”
- `<target name="info">`
  and its closing tag `</target>` specify what to do if “ant info” (or because info is default, simply “ant”) is typed at the command line
- `<echo>Hello, World!</echo>`
  specifies that the string “Hello, World!” should be printed (echoed) to the console

Hello, World in Ant

- The following is “Hello, World!” in Ant

```
<?xml version="1.0"?>
   <project name="Hello World Project" default="info">
   <target name="info">
      <echo>Hello, World!</echo>
   </target>
</project>
```

```
ricegf@antares:~/dev/202108/Examples/Ant/hello$ ant
Buildfile: /home/ricegf/dev/202108/Examples/Ant/hello/build.xml

info:
     [echo] Hello, World!

BUILD SUCCESSFUL
Total time: 0 seconds
```

be

simply “ant”) is typed at the command line

- `<echo>Hello, World!</echo>`
  specifies that the string “Hello, World!” should be printed (echoed) to the console

A Slightly More Practical Ant

- Use this file to specify how to build Java applications

build.xml

```
<?xml version="1.0"?>
<project name="CSE1325" default="build">

  <target name="build" description="Compile source tree java files">
    <javac includeantruntime="false" debug="true" failonerror="true">
      <src path="."/>
    </javac>
  </target>

  <target name="clean" description="Clean output files">
    <delete dir="docs/api"/>
    <delete>
      <fileset dir=".">
        <include name="**/*.class"/>
      </fileset>
    </delete>
  </target>
</project>
```

Type ‘ant’ to build your code

>> Type ‘ant clean’ to delete all .class files and start over

Include this build.xml file in ALL git homework

>> Graders will ‘**`ant clean ; ant`**’ to build & test your code

What Targets Are Defined?

- make famously can’t tell you which targets are defined in your Makefile
  - We have Python for that
- Ant can do this with the -p option
  - See ant -h for many, many other options!

```
ricegf@antares:~/dev/202308/P01/full_credit$ ant -p
Buildfile: /home/ricegf/dev/202308/P01/full_credit/build.xml

Main targets:

 build  Compile source tree java files
 clean  Clean output files
Default target: build
ricegf@antares:~/dev/202308/P01/full_credit$
```

More?

- See ant -h for many, many other options!

```
ricegf@antares:~/dev/202308/P01/full_credit$ ant -h
/usr/bin/ant [script options] [options] [target [target2 [target3] ..]]
Script Options:
  --help, --h            print this message and ant help
  --noconfig             suppress sourcing of /etc/ant.conf,
                         $HOME/.ant/ant.conf, and $HOME/.antrc
                         configuration files
  --usejikes             enable use of jikes by default, unless
                         set explicitly in configuration files
  --execdebug            print ant exec line generated by this
                         launch script

ant [options] [target [target2 [target3] ...]]
Options:
  -help, -h              print this message and exit
  -projecthelp, -p       print project help information and exit
  -version               print the version information and exit
  -diagnostics           print information that might be helpful to
                         diagnose or report problems and exit
  -quiet, -q             be extra quiet
  -silent, -S            print nothing but task outputs and build failures
  -verbose, -v           be extra verbose
  -debug, -d             print debugging information
  -emacs, -e             produce logging information without adornments
  -lib <path>            specifies a path to search for jars and classes
  -logfile <file>        use given file for log
    -l     <file>                  ''
  -logger <classname>    the class which is to perform logging
  -listener <classname>  add an instance of class as a project listener
  -noinput               do not allow interactive input
  -buildfile <file>      use given buildfile
    -file    <file>              ''
    -f       <file>              ''
  -D<property>=<value>   use value for given property
  -keep-going, -k        execute all targets that do not depend
                         on failed target(s)
  -propertyfile <name>   load all properties from file with -D
                         properties taking precedence
  -inputhandler <class>  the class which will handle input requests
  -find <file>           (s)earch for buildfile towards the root of
    -s  <file>           the filesystem and use it
  -nice  number          A niceness value for the main thread:
                         1 (lowest) to 10 (highest); 5 is the default
  -nouserlib             Run ant without using the jar files from
                         ${user.home}/.ant/lib
  -noclasspath           Run ant without using CLASSPATH
  -autoproxy             Java1.5+: use the OS proxy settings
  -main <class>          override Ant's normal entry point
ricegf@antares:~/dev/202308/P01/full_credit$
```

Do Professionals Use Ant?

- Sometimes, but not usually by itself
  - Ant integrates with the Gradle project management tool for more sophisticated features
    - Faster, better, cleaner
    - More features like incremental / triggered build
  - Maven is the most common build automation framework for Java
    - Uses extensive conventions to simplify builds
    - Independent of but similar to Ant (e.g., XML build files)
- Learning Ant will prepare you to use more sophisticated tools later on

Git Conflict / Merge Example

When pushing or pulling, you *may occasionally* see a CONFLICT message.

```
ricegf@pluto:~/dev/cpp/cse1325/P01/bonus$ git pull
warning: no common commits
remote: Enumerating objects: 51, done.
remote: Counting objects: 100% (51/51), done.
remote: Compressing objects: 100% (43/43), done.
remote: Total 51 (delta 8), reused 35 (delta 3), pack-reused 0
Unpacking objects: 100% (51/51), done.
From https://github.com/prof-rice/cse1325
 + aef0967...cb7b58a master     -> origin/master  (forced update)
Auto-merging README.md
CONFLICT (add/add): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.
ricegf@pluto:~/dev/cpp/cse1325/P01/bonus@ cat README.md
```

This just means that a file's content was modified on both your laptop AND on GitHub, so git isn't sure which content to keep.

What to do? Merge!

https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/addressing-merge-conflicts/resolving-a-merge-conflict-using-the-command-line

If you have trouble, contact the TA. They're experts!

Review

[rejected] !!! Pull then Push

- If a push is rejected
  - Pull first
  - Push again!
- If that fails, clone again! (see next slide)

```
ricegf@antares:~/dev/202108/QB/study_sheet$ git add -u
ricegf@antares:~/dev/202108/QB/study_sheet$ git commit -m 'E2 study sheet update'
[main 6523776] E2 study sheet update
 3 files changed, 78 insertions(+), 52 deletions(-)
 rewrite QB/study_sheet/widgets.png (98%)
ricegf@antares:~/dev/202108/QB/study_sheet$ git push
To https://github.com/prof-rice/202108.git
 ! [rejected]        main -> main (fetch first)
error: failed to push some refs to 'https://github.com/prof-rice/202108.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
ricegf@antares:~/dev/202108/QB/study_sheet@ git pull
remote: Enumerating objects: 17, done.
remote: Counting objects: 100% (17/17), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 9 (delta 6), reused 9 (delta 6), pack-reused 0
Unpacking objects: 100% (9/9), 2.25 KiB | 1.13 MiB/s, done.
From https://github.com/prof-rice/202108
   6fa3989..fbed158  main       -> origin/main
Merge made by the 'recursive' strategy.
 P07-JADE/tags/Makefile             |  2 +-
 P07-JADE/tags/sprint2.tags         | 81 ++++++++++++++++++++++++++++++++++++------
 P08-JADE/full-credit/Untitled.jade | 22 +++++++-------
 3 files changed, 67 insertions(+), 38 deletions(-)
ricegf@antares:~/dev/202108/QB/study_sheet$ git push
Enumerating objects: 21, done.
Counting objects: 100% (16/16), done.
Delta compression using up to 12 threads
Compressing objects: 100% (9/9), done.
Writing objects: 100% (9/9), 114.86 KiB | 2.17 MiB/s, done.
Total 9 (delta 6), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (6/6), completed with 6 local objects.
To https://github.com/prof-rice/202108.git
   fbed158..996fc66  main -> main
ricegf@antares:~/dev/202108/QB/study_sheet$
```

Review

[rejected] !!! – Clone Again

- **Rename** your cse1325 directory to cse1325-bad.
- **Clone** your repository again, which creates a new working cse1325 directory.
- **Copy** the files you want to push from cse1325-bad into cse1325.
- In cse1325, **add, commit, & push** the files. Life is good. Back to the homework!
- When you're sure you have all the files you need from cse1325-bad, **delete** it like a bad dream.

```
ricegf@antares:~$ lt cse1325/
Permissions Size User   Date Modified Name
drwxrwxr-x     - ricegf 07-20 17:21   cse1325
drwxr-xr-x     - ricegf 07-20 17:21   └── P01
drwxr-xr-x     - ricegf 07-20 17:21       ├── bonus
.rw-rw-r--   287 ricegf 07-20 17:21       │   ├── Hello.java
.rw-rw-r--  132k ricegf 07-20 17:21       │   ├── Hello1.png
.rw-rw-r--  136k ricegf 07-20 17:21       │   └── Hello2.png
drwxr-xr-x     - ricegf 07-20 17:21       ├── extreme_bonus
.rw-rw-r--   174 ricegf 07-20 17:21       │   ├── Hello.java
.rw-rw-r--   74k ricegf 07-20 17:21       │   ├── Hello1.png
.rw-rw-r--  115k ricegf 07-20 17:21       │   └── Hello2.png
.rw-rw-r--   73k ricegf 07-20 17:21       ├── file_hierarchy.png
.rw-rw-r--   60k ricegf 07-20 17:21       ├── file_structure.png
drwxr-xr-x     - ricegf 07-20 17:21       └── full_credit
.rw-rw-r--   131 ricegf 07-20 17:21           ├── Hello.java
.rw-rw-r--   64k ricegf 07-20 17:21           └── Hello.png
ricegf@antares:~$ mv cse1325/ cse1325-bad/
ricegf@antares:~$ git clone https://github.com/prof-rice/cse1325.git
Cloning into 'cse1325'...
remote: Enumerating objects: 10, done.
remote: Counting objects: 100% (10/10), done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 10 (delta 1), reused 5 (delta 0), pack-reused 0
Receiving objects: 100% (10/10), 13.02 KiB | 459.00 KiB/s, done.
Resolving deltas: 100% (1/1), done.
ricegf@antares:~$ lt cse1325
Permissions Size User   Date Modified Name
drwxrwxr-x     - ricegf 07-20 17:22   cse1325
.rw-rw-r--    35k ricegf 07-20 17:22   ├── LICENSE
drwxrwxr-x     - ricegf 07-20 17:22   ├── P01
drwxrwxr-x     - ricegf 07-20 17:22   │   └── full_credit
.rw-rw-r--    13 ricegf 07-20 17:22   │       └── Hello.java
.rw-rw-r--     9 ricegf 07-20 17:22   └── README.md
ricegf@antares:~$ # copy important files from cse1325-bad to cse1325
```

Your ID here!

How Often to Commit to GitHub?

- Commit every 15 minutes or so
  - You added a new feature
  - You fixed a bug
  - Your code compiles but breaks, and you’re about to make a lot of changes to “fix” it
  - You wrote *more than you want to rewrite*
- When in doubt – commit!

What Goes into git (and what doesn't)

- Include in your git repository
  - Source code (e.g., *.java, *.cpp, *.h)
  - Custom scripts and project-unique tools (build.xml, Makefile, configure)
  - UML models (e.g., *.uml, *.xmi)
  - Runtime artifacts (e.g., toolbar icon image files, data files)
  - Documentation that is specific to each release (e.g., README.md, release notes)
- Omit from your git repository
  - Build artifacts (e.g., *.class, executables, and .o files) – consider a continuous integration server or a separate git instance, if needed
    - Host Javadoc on https://javadoc.io/ or your GitHub project pages
  - Standard tools (e.g., OpenJDK, gcc) – rely on web sources (e.g., apt)
  - Generic project documentation (e.g., *.docx) – consider a wiki
  - Bug reports – consider a bug tracker (e.g., GitHub Issues, Jira)

What We Learned Today

- Some differences between C and Java
  - Types and declarations
  - Console I/O
  - Syntax
- Packages and imports
- Building with Ant
- A bit more git

For Next Class

- Review the slides, code, and reading material from today
- Take the “Lecture 01 Quiz” on Canvas (5 minutes, tops!)
- Complete assignment P01 and push to GitHub for grading
- Skim the Lecture 03 material in advance (it posts the day before)
- **Watch Lecture 02** (video only in Modules) by Lecture 04
  - NOT optional! Video log is the pop quiz grade
  - This has a required “Lecture 02 Quiz” as well

LECTURE VIDEOS

Since this is a bonus lecture, a pop quiz grade will not be taken. However, the material covered (additional examples for Lecture 01) in inherently on the exam and the post-lecture quizlet will be graded.

Chapter 01 - Multiplication Table