

Java 100/105

procedural coding



Java 100/105 course structure



- 5 days
- Java 100
 - For non-programmers
 - Thorough programming instruction
- Java 105
 - For programmers / Java 100 students
 - Review of programming
 - Focus on OO

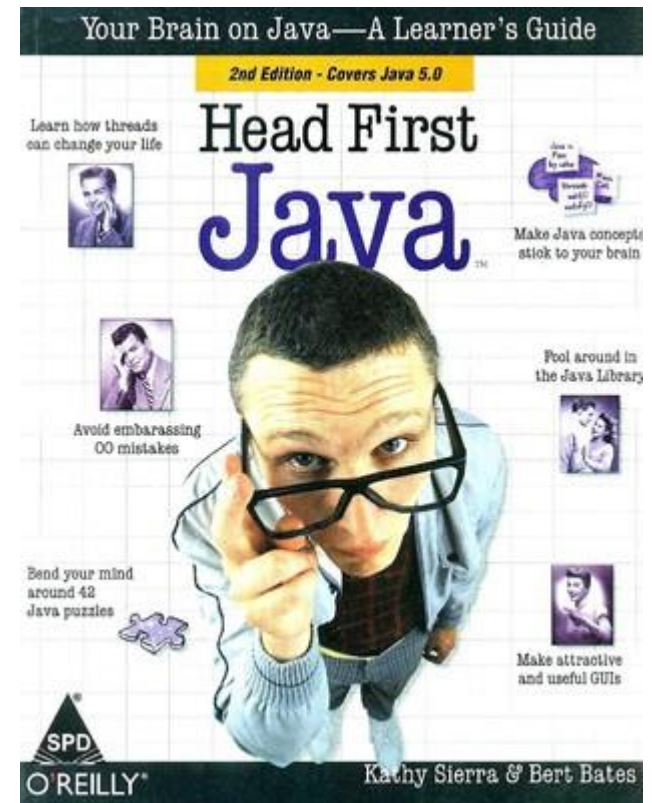


Books - required

- Required for 100
 - **Beginning Programming With Java For Dummies**
- Required for 105
 - **Java: A Beginner's Guide**, 5th Ed., Herbert Schildt, McGraw Hill, Aug 2011, \$40
- Required for 200
 - **Java The Complete Reference**, 8th Ed., Herbert Schildt, McGraw Hill, Jun 2011, \$60

Books - recommended

- O'Reilly's Head First series
 - Head First Java
 - Head First Servlets & JSPs
 - Head First Design Patterns





Web site resources

- <http://github.com/doughoff>
- [Java](#)
 - download either 32 or 64 bit Java 8 SE JDK if you aren't using an IDE
- Text editors
 - [Notepad++](#) - simple, popular



Tools - IDEs

- [Eclipse](#) – open source
 - EE for web sites
 - Classic for smaller install, smaller menus
- [jDeveloper](#) – Oracle
- IntelliJ – JetBrains \$
- WAD (Websphere App Dev) – IBM \$
 - RAD (Rational App Dev)



Download

- Get **Java 9** 64-bit JDK (SE works fine)
 - Search for “Java 9 download”
- Get the **Eclipse** IDE for Java **EE** Developers 64-bit ()
 - Neon – June 2016
 - Oxygen - now



History, comparisons

LANGUAGE BACKGROUND



Origins

- C, C++
- Interactive TV, networked devices
- Internet
- 1991, public release 1.0a2 1995
- Managed by open source Java Community Process since 1.4



Transitions

- Sun Microsystems developed
- Netscape partnered to embed Java engine in browser.
- Microsoft lawsuit ended browser possibilities
 - Sun originally sued against J++
 - Microsoft forced users to download IE plug-in
 - Sun forced Windows to have JRE
- Sun open-sourced all Java code
- Sun bought MySQL



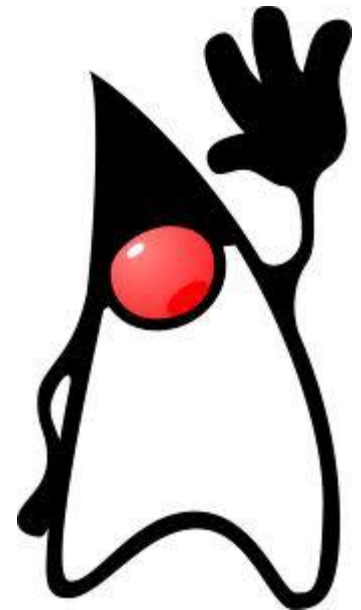
New owner

- Sun purchased by Oracle
 - IBM was a possible buyer
- Oracle uses Java extensively
 - Back end integration (Financials, Fusion)
- Oracle and MySQL
 - An upgrade path – free without support



Duke

- The Java mascot
- born May 23, 1995,
the first demo of Java
technology publicly
released.
- Updated each year for a theme
 - JavaOne – Sep, 201?,
San Francisco





Similar languages

- C++ - procedural & OO
- C#
 - Microsoft's revised version of J++
 - Has been adding features while Java was laying low.
 - 80% of keywords/syntax borrowed from Java
- JavaScript
 - executes in and manages the browser
 - several libraries copied
 - 70% of keywords/syntax borrowed from Java



Java structure - SE

- Java SE – standard edition
 - Java 1.1
 - J2SE 1.2, 1.3, 1.4, 5.0
 - Java SE 6, 7, 8, 9
- Some packages moved out of or into SE over time.
- Represents most common usage for console / GUI / middleware apps



Java structure - EE

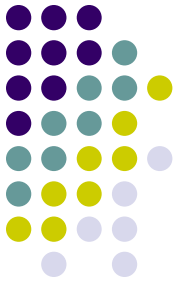
- Java EE – enterprise edition 1998
 - J2EE -> Java EE 5, 6
 - Web – servlets, applets, JSP, JSF, Tomcat (web server)
 - Networking, database (persistence, JDBC), email, web services, XML, components (EJBs)
 - Typically used with application servers
 - A web server with additional features
 - GlassFish, JBoss, Oracle WebLogic, IBM WebSphere



Development process

- Write code in text file
 - source code
 - file name ends with **.java**
- Compile source code
 - compiles to bytecode, binary machine language
 - file name ends with **.class**
 - builds (javac.exe) with Java's code libraries
- Move bytecode to destination platform
- Run bytecode (java.exe)

JDK

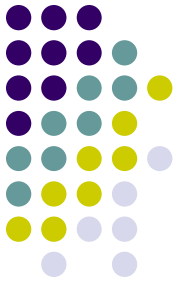


- Java Development Kit
 - JRE – Java Runtime Environment
 - java.exe – runs bytecode, different for each platform
 - APIs – Java code libraries
 - javac.exe – compiles source code
 - javadoc.exe – makes html documentation from source code with special comments

JVM



- or just VM (virtual machine)
- the engine that executes the code
 - java.exe
 - The sandbox



Development environment

ECLIPSE OXYGEN



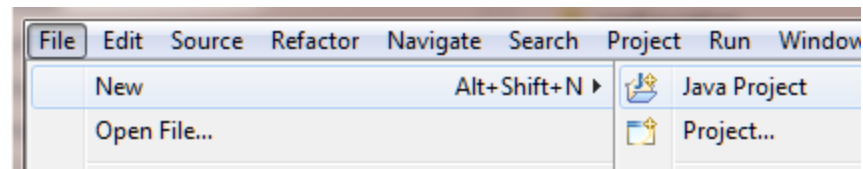
Eclipse

- IBM donated code to public domain from VisualAge
- IBM now maintains versions as Rational Application Developer (RAD) for the WebSphere suite (WAD)
 - WebSphere is the brand used for server products
- Many plug-ins allow feature additions
- Uses SWT and not Swing for GUI
 - SWT is simpler and preferred for GUI dev

Projects



- Workbench – the development area
 - Project explorer – logical view of your files, not Windows Explorer view
- Project
 - Eclipse file information
 - Run configurations
- File / New / Java Project



Class project

- Project name
- Location
 - for backups
- JRE
 - for older environments
- Separate folders
 - for zips/jars to server

Create a Java Project

Create a Java project in the workspace or in an external location.



Project name:

☒ Use default location

Location: [Browse...](#)

JRE

☒ Use an execution environment JRE:

☐ Use a project specific JRE:

☐ Use default JRE (currently 'jre6') [Configure JREs...](#)

Project layout

☐ Use project folder as root for sources and class files

☒ Create separate folders for sources and class files [Configure default...](#)

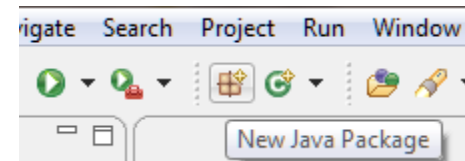
Working sets

☐ Add project to working sets

Working sets: [Select...](#)

Packages

- Categories of files
- Creates a directory to hold files
- Lower case first letter



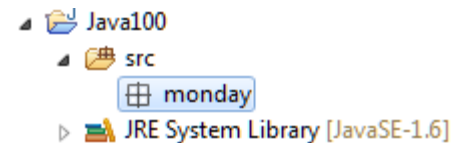
Java Package

Create a new Java package.

Creates folders corresponding to packages.

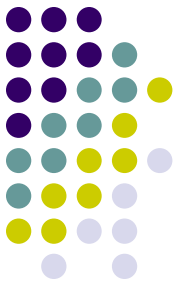
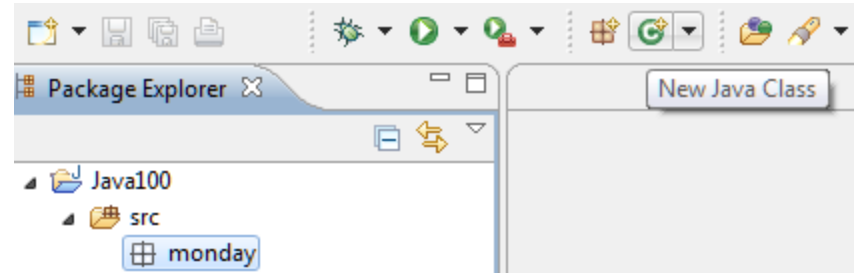
Source folder: Java100/src

Name: monday



Class creation

- Modules of code
- Highlight package
- Click icon





Class configuration

- Confirm package
- Name starts with a capital letter

Java Class

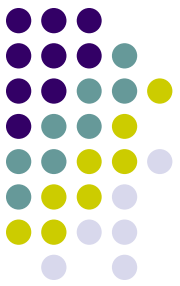
Create a new Java class.

Source folder:	Java100/src
Package:	monday
<input type="checkbox"/> Enclosing type:	

Name:	HelloWorld
Modifiers:	<input checked="" type="radio"/> public <input type="radio"/> default <input type="radio"/> private <input type="radio"/> protected <input type="checkbox"/> abstract <input type="checkbox"/> final <input type="checkbox"/> static
Superclass:	java.lang.Object
Interfaces:	

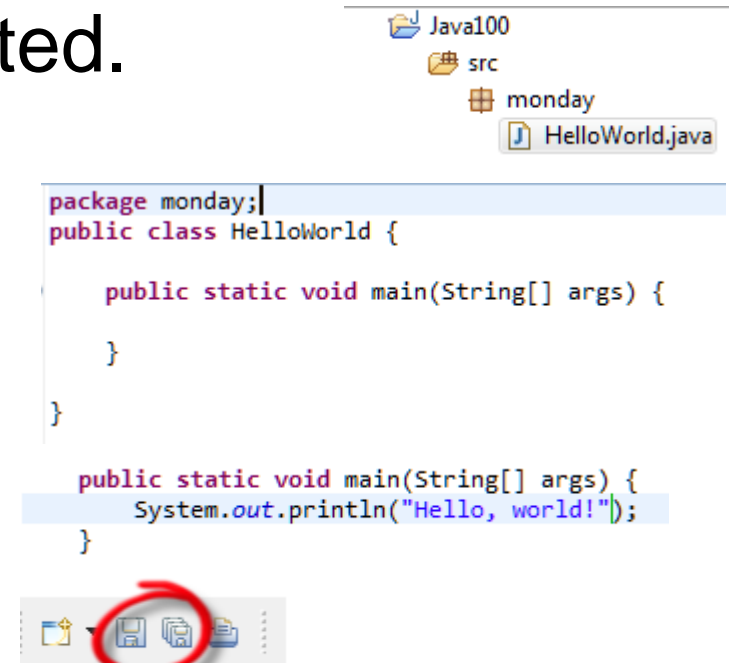
Which method stubs would you like to create?

☒ public static void main(String[] args)
☐ Constructors from superclass
☒ Inherited abstract methods



Class code

- Comments can be deleted.
 - `/** comment */`
 - `// comment`
- Type code in `main()` {
}
- `System.out.println("Hello, world!");`
- **Save** or **Save All** to build
 - errors will show at bottom



The screenshot shows an IDE window with a project named 'Java100'. Inside the 'src' folder, there is a package named 'monday' containing a file 'HelloWorld.java'. The code in the editor is as follows:

```
package monday;
public class HelloWorld {

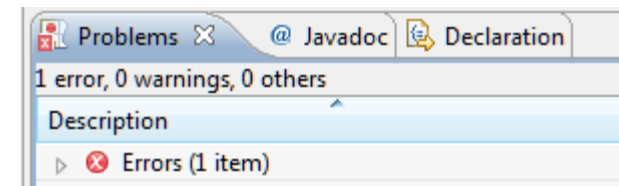
    public static void main(String[] args) {

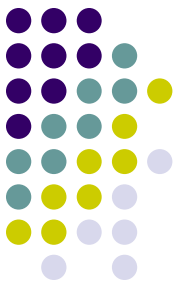
    }

}

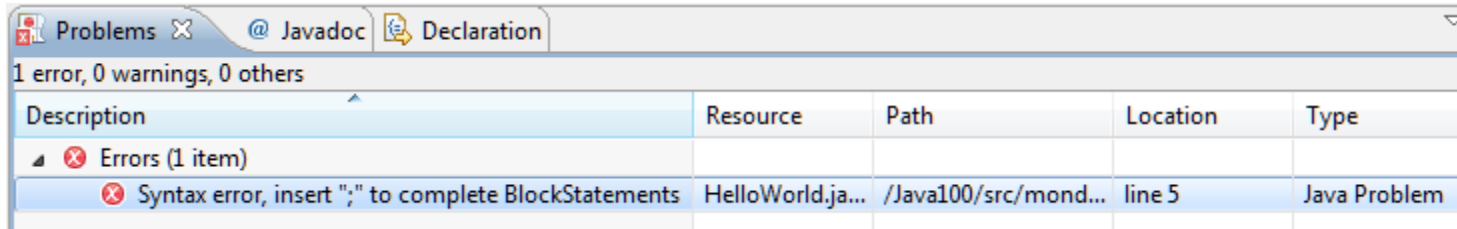
public static void main(String[] args) {
    System.out.println("Hello, world!");
}
```

At the bottom of the editor, there is a toolbar with icons for 'New', 'Open', 'Save', 'Save All', and 'Run'. The 'Save' icon (a floppy disk) is circled in red.





Error fixing

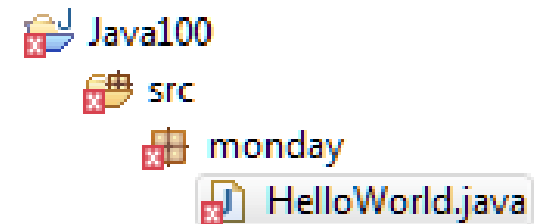


Problems | @ Javadoc | Declaration

1 error, 0 warnings, 0 others

Description	Resource	Path	Location	Type
▲ [X] Errors (1 item)				
[X] Syntax error, insert ";" to complete BlockStatements	HelloWorld.java...	/Java100/src/mond...	line 5	Java Problem

- Double click error to go to line
- Error icons will show from class up to project until built
- Error icon on left ribbon is at line
 - Grays out when fixed before build
- Error icon on right ribbon shows all errors in file



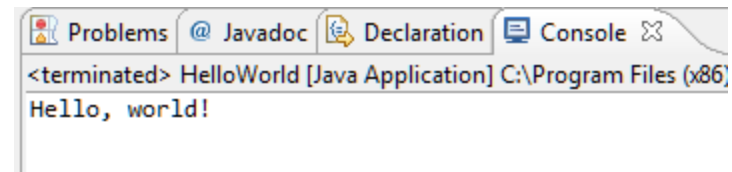
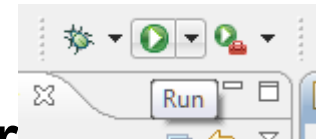
```
System.out.println("Hello, world!");  
}
```



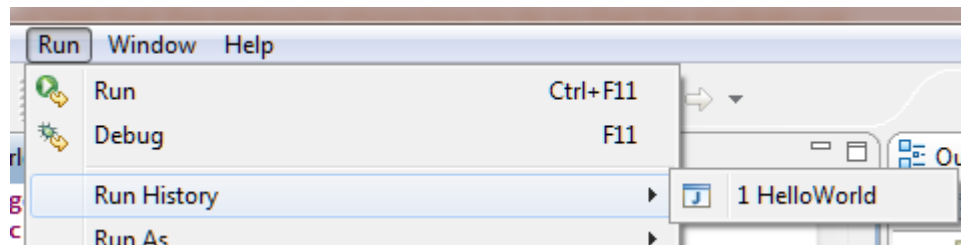


Class execution

- Click icon
- **Console** will appear with your output



- Configuration was saved in **Run History** and can be run anytime





Simple keywords: if, for, code blocks...

SYNTAX



Code blocks

- Any group of statements between { }
- Can be nested
 - {
 - { }
 - } // end of ?
- Used to organize code as one unit



Statement

- One line of Java code
- Almost always enclosed in a code block
 - package
 - import
- Always ends in a semicolon;
- Use as much white space as you want



Statement types

- Value assignment
 - direction is right to left
 - `y = 5;`
 - `z = 0;`
 - `x = y;`
 - `y = z;`
- Execute a method
 - `System.out.println("Hello, world!");`



Keywords

- Words reserved by Java
- Colored in purple

```
package monday;  
public class HelloWorld {  
    public static void main(  
        System.out.println("Hello World!");  
    }  
}
```



Identifiers

- Names you define for variables, class names, package names, method names
- Remains in black
- Rules
 - first letters are a-z, A-Z, \$, and _
 - _ is not valid by itself in SE 8
 - no spaces
 - hardly any length limit
 - case sensitive

```
int x = 5;
```



Java style

- Class names
 - first letter capitalized, no `_`, first letter of any word is capitalized
 - ThisIsALongClassNameExample
- Method names, variables, packages
 - first letter is not capitalized , first letter of any word is capitalized (Camel case)
 - thisIsAnExampleOfALongVariableName



Java APIs

- The pre-written class libraries supplied by Oracle
- Versioned
- Needs to be declared before use
 - except for `java.lang` classes
- Search for “[Java API docs](#) 9”



Order of execution of process steps

PROGRAM FLOW 1



Program flow

- What controls the execution of one task after another task.
 - A to do list
 - A fork in the road
 - A TV channel selector
 - Wash, rinse, repeat.
 - A task given to many people at the same time



Program flow types

- Script
 - Execute each line until end of main().
- Units
 - Execute reusable modules of code in another place as needed.
- Controller
 - Call units of code to do all the work.



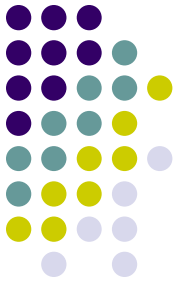
Program flow types

- Branching
 - Check the state of a rule (test) and execute code based on results
- Iteration
 - Execute a unit of code multiple times all at once until a rule is met (test).



Program flow types

- Asynchronous calls
 - Execute a unit of code anytime in the near future and get a reply whenever.
 - Overlapping
 - Two types
 - Distributed – AJAX
 - Non-distributed - Threading



the for loop, while, do-while

LOOPS 1



Loops / Iteration control

- A loop is the syntax used to repeat a set of statements to
 - Execute a task on each one of many things
 - Continue to do something until a limit is reached
 - Do something forever like wait for a request to send something (web server)



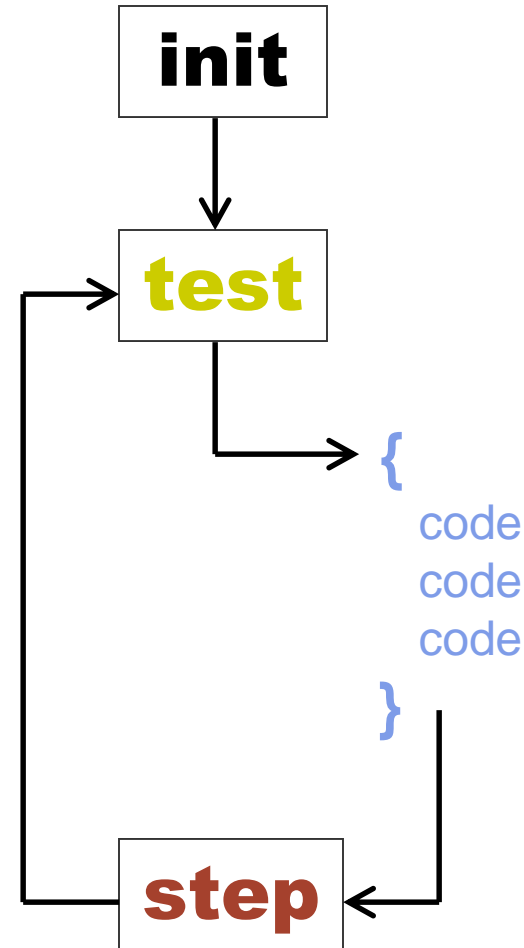
Java loop syntax types

- for
- while
- do-while
- for each (enhanced for)
- functional style for with streams



for loop logic

- **initialization** before 1st iteration
- conditional **testing**
- code block or statement iterations
- “**stepping**” statements



for syntax



```
for ( init ; test ; step ) {  
    code  
    code  
    code  
}
```



for syntax - init

for (**init** ; **test** ; **step**)

- local variables declared and initialized
- multiple variables separated by commas
- can be empty
- Examples:
 - `int i = 0`
 - `int i = 0, j = 1`
 - `Iterator iterator = list.iterator()`



for syntax - test

for (init ; test ; step)

- result is true or false
- occurs before each iteration
- false result goes to end of code block
- Examples:
 - `i <= 10`
 - `i < array.length`
 - `iterator.hasNext()`



for syntax - step

for (init ; test ; step)

- occurs at the end of each iteration
- Examples:
 - $i = i + 1$
 - $i++$
 - $j = i * 2$



for loop 1

```
public class Count {  
    public static void main (String[ ] args) {  
        for ( int i = 0; i < 10 ; i=i+1 ) {  
            System.out.println("i = " + i);  
        }  
    }  
}
```



for loop 2

```
public class Count {  
    public static void main (String[ ] args) {  
        for ( int i = 0; i >= 10 ; i=i+1 ) {  
            System.out.println("i = " + i);  
        }  
    }  
}
```



for loop 3

```
public class Count {  
    public static void main (String[ ] args) {  
        for ( int i = 0; i <= 10 ;    )  
            System.out.println("i = " + i);  
            i = i + 1;  
    }  
}
```



for loop 4

```
public class Count {  
    public static void main (String[ ] args) {  
        for ( int i = 0; i <= 10 ; ) {  
            System.out.println("i = " + i);  
            i = i + 1;  
        }  
    }  
}
```



for loop 5

```
public class Count {  
    public static void main (String[ ] args) {  
        int i = 0;  
        for (    ; i <= 10 ; i=i+1 ) {  
            System.out.println("i = " + i);  
        }  
        // i is usable here  
    }  
}
```

...



for turned into a while

```
public class Count {  
    public static void main (String[ ] args) {  
        int i = 0;  
        while ( i <= 10) {  
            System.out.println("i = " + i);  
            i=i+1;  
        }  
    }  
}
```

...



for loop summary

- most used iterative control
- used for counting
- used for operations on each element of an array or collection
- only control that uses variable declaration
- semicolon operator & parentheses for grouping 3 parts are unique to **for**



Exercise

- Print out a list of numbers that:
 - Print from 1 to 25
 - Print from 25 to 1
 - Print from 1 to 50 by 2's
 - Print from 1 to 10 by .5



Loop keywords - structure

- **while**

- like **for** loop but no **init** or **step**
- only declares a test
- do **init** above loop, **step** as last statement in loop.

- **do-while**

- one iteration guaranteed
- then, same as while loop



Ending code

- Statements end in a semi-colon
- Loops end in a code block with a curly brace pair



Exercises (PowerTests)

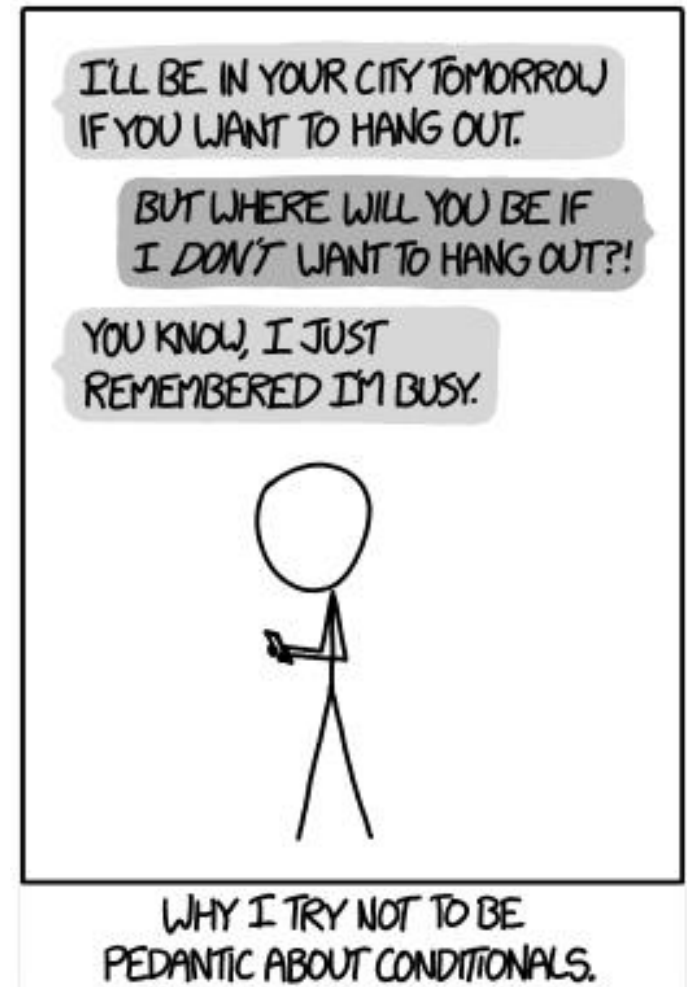
- Use a loop to print the powers of 2 from 0 to 50.
 - $2^0 = 1$
 - $2^1 = 2$
 - $2^2 = 4$
 - $2^3 = 8 \dots$
 - Use a long variable to store your results if int fails
 - For an advanced version, use `Math.pow(number,power)`



If you're done being pedantic, we should get dinner.'
'You did it again!' 'No, I didn't.'

the if and switch statements

BRANCHING





Branching types

- 1-way
 - if-then
- 2-way
 - if-then
 - else
- 3-way
 - if-then
 - else if-then
 - else
- 4-way...



Branching types

- Multiple branches
 - switch – case 1, case 2, case 3...
- Branching with events
 - tie code to any event



Conditionals

- **if** (this is true) execute this statement;
- **if** (this is true) {
 execute this entire code block;
}
- **if** (this is true)
 { execute this block; }
else
 { execute this block; }



Loop keywords – test in loop

- Used with condition/rule
- **break**
 - halt all iterations of loop; go to end of code block
 - abort
- **continue**
 - like break but will start on the next iteration of the loop
 - skip this iteration



A real life bug in iOS

- Goto fail bug existed Sep 2012 - Feb 2014
- C
- ```
if ((err =
 SSLHashSHA1.update(&hashCtx,
 &signedParams)) != 0)
 ● goto fail;
 ● goto fail;
 ● more code
```



# Nested ifs

- if ( condition 1 ) {
  - true condition 1
- }
- else {
  - if (condition 2) {
    - true condition 2 and false condition 1
  - }
  - else { false condition 1 and false condition 2 }
- }



## Exercise (NestedIfs)

- Set up three print statements for matching one value of either 1, 2, or 3.
  - if ( i ==1) { System.out.println("1"); ...



# switch

- a multiple result check for **one** test
  - switch (expression or test) {
  - case expressionA:
  - statements; break;
  - case expressionB:
  - statements; break;
  - default:
  - statements;
  - }



# switch

- switch on whole numbers
  - byte, short, int, long, char
  - enums
  - Strings in 1.7
- Use break; to stop execution for each case
- Don't use break to group cases (fall-through or empty cases)
- **Eclipse**: type switch, Ctrl-spacebar



# Exercise

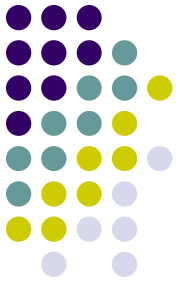
- Set up a unique print statement for any value from 1 to 5.
- Set up a unique print statement for any value under 10, between 10 and 20 and over 20 using a switch.



# Exercise (LoopTests)

- Print out numbers 0 to 99
  - Formatted to look like:
    - 0 1 2 3 4 5 6 7 8 9
    - 10 11 12 13 14 15 16 17 18 19
    - 20 ...
  - One loop only





Working with variables

# DATA TYPES



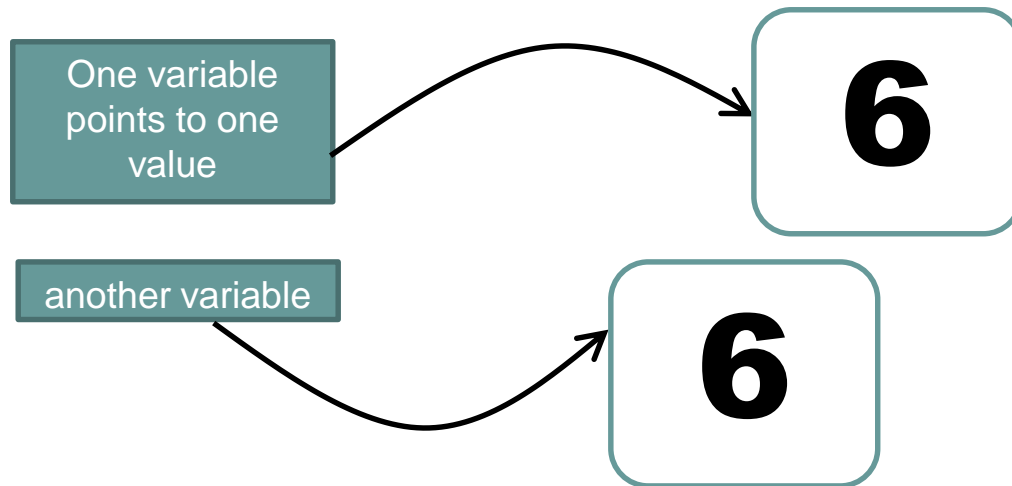
# Introduction

- A data type
  - Constrains min/max if **whole number**
  - Constrains significant digits if **floating point**
  - Constrains data structure if **class**
  - Constrains data indices and structure if **array**
- The JVM
  - Checks for validity when building (compile time check)



# Value types

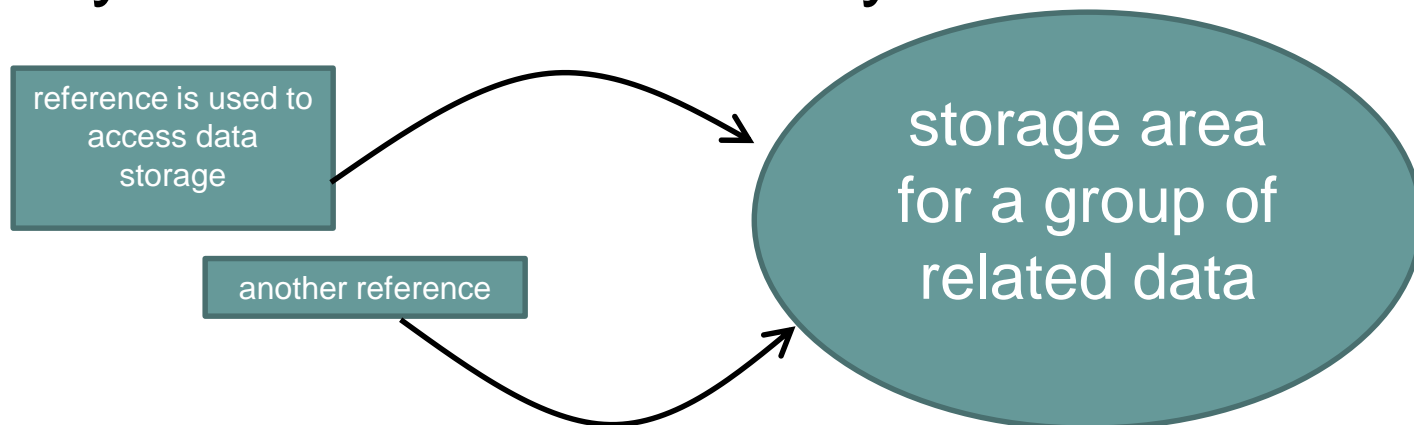
- Two types – value (primitive) and reference
- Values point to stack storage.





# Reference types

- Variables refer to a storage area. They NEVER refer to other references.
- Storage area must be created first.
- Absence of a storage area means reference points to **null**
- Any reference can modify data.





# Value (primitive) types

- Whole numbers
  - **byte**, **short**, **int**, **long**
- Floating point
  - **float**, **double**
- Logical
  - **boolean** (**true**, **false**)
- **Char**
  - any one character
  - literals must be enclosed in single quotes: 'a'
  - really a **short** that is only positive (unsigned)



# Basic steps of variable use

- Declare a variable
  - datatype and identifier
  - only declared once
- Reserve the memory for the data
  - only for references
- Initialize the values
  - local variables must be initialized before use



# Declaring variable names

- Datatype must be declared only once
  - `int i;`
  - `double d;`
- Variable will then be compile-time checked.



# Initializing variables

- Reserving memory for values is automatic.
- Literals or other variables of the same type are assigned to variables with =
  - `int i;`
  - `i = 0;`
- Declaring and initializing can be combined
  - `int i = 0;`





# Default types for literals

- Whole number literals are **ints**
  - `int i = 128;`
  - `byte b = 128;` (out of range)
- Floating point number literals are **doubles**
  - `double d = 128.0;`
  - `float f = 128.0` (can not change datatype)



# Other number formats

- Hex – starts with 0x
  - 0xABCDEF123
- Octal – starts with 0
  - 076543
- Scientific notation – ends with e and exponent for base 10 multiplier
  - 1.234567e22



# Casting

- changing the data type
- place the new datatype in ( ) before the value
  - `byte b = (byte) 128;`
  - `float f = (float) 128.0;`
- Shortcuts
  - `float f = 128.0f;`
  - `float f = 128.0F;`
  - `double d = 5d;`



# Floating point inaccuracy

- Create a new class FunnyMoney with a main( ) to run this:
- `double dollar = 1.00;`
- `double dime = dollar/10;`
- `double total = 0;`
- `total = dime + dime + dime + dime + dime +  
dime + dime + dime + dime + dime;`
- `System.out.println(total);`



# Currency

- Don't use a double for currency
  - use a rounding strategy if you do, be careful
- Use BigDecimal class
  - Also available is a BigInteger class for higher values.



# Scope

- Variables declared in a code block are only visible (usable) in that code block or nested code blocks, not outside.
  - `int i = 0;`
  - `{ int j = i ; j++; }`
  - `System.out.println(j);`
- Also called lifetime or visibility
- Local variables live only in the code block when declared.



# Exercise (DivisionTests)

- What is the output?
  - `System.out.println(1/1);`
  - `System.out.println(1/0.0);`
  - `System.out.println(1/2);`
  - `System.out.println(2/3);`
  - `System.out.println(2/3.0);`



# Text

- Allows for both value and reference type forms of creation.
- Uses the **String** datatype
  - `String s = "text";`
- Stores text in Unicode not ASCII
  - Output to console is based on platform's native character set





# String variables

- Almost like a value type
- Can be declared and initialized at once
- Create variables like primitives
  - `String s = "Text";`



# String literals

- Uses double quotes to delimit value
  - “dog”
- Can use concatenation operator
  - “Rover the ” + “dog”
  - will convert any thing to text if adding to text
  - 1 + 2 + “3” + 4 + 5
  - “” + 10 + 11
- Escape sequences use \
  - “1\n2”, “1\t2” , “c:\\users\\doug”



# Exercises (CharTests)

- What is the output?
  - `System.out.println('A');`
  - `System.out.println(0 + 'A');`
  - `System.out.println( (char)(1 + 'A') );`
  - `System.out.println( 'A' + 1);`
  - `System.out.println('A' + 'B' + 'C');`
  - `System.out.println("A" + "B" + "C");`
  - `System.out.println("Alphabet: " + 'A' + 'B' + 'C');`



Symbols that take action

# OPERATORS



# Arithmetic

- **+, -, \*, /**
- **% modulus**
  - 5 % 3
  - 15 % 4
  - 5 % 9
- **++, --**
  - X++ , ++X
  - `int x = 0;`
  - `System.out.println(x++);`
  - `System.out.println(++x);`



# Relational

- `==` is equal to?
- `!=` is not equal to?
- `>` , `<`, `>=`, `<=`



# Logical

- Any TRUE in a OR (union) expression makes result TRUE
- TRUE | TRUE | FALSE | TRUE | TRUE
- Any FALSE in an AND (intersection) expression makes result FALSE
- TRUE & TRUE & FALSE & TRUE & TRUE



# Logical

- Bit-wise
  - will execute always
  - `&` and, `|` or, `^` xor, `!` Not
  - mostly for graphics, subnetting
- Short-circuit
  - will stop if useless to continue
  - `&&` and, `||` or
  - `(true || false || false || true)`
  - `(false && true && false && true)`





# Other

- Assignment =
- Compound
  - **`+=`, `-=`, `*=`, `/=`, `%=`**
  - **`x = x + 1` or `x++` or `x += 1`**
  - **`x = x + 2` or `x += 2`**
  - **`String s = "a"; s += "bc";`**
  - `x+=10`
  - `x*=.90`



# Operator precedence

- Order in which operator is executed when other operators are present.
- Directional
- Don't memorize, use parentheses
  - $1 + 2 * 3 + 4$
  - $(1+2) * (3+4)$



# If-else replacement (ternary op)

- (expression) **?** value if true **:** value if false
- `int policies = 0;`
- `System.out.println(policies + " polic" +  
( (policies ==1) ? "y" : "ies" ) ); // handle plural`
- Results
  - 0 policies
  - 1 policy
  - 2 policies



# Exercise – desk check

- `boolean needsCookie = false;`
- `int time = 2;`
- `(5 == 6) || (true != false)`
- `(5 == 6) && (true != false)`
- `(time >= 2) || needsCookie || --time == 1`
- `(time >= 2) && !needsCookie`



# Exercise – (OperatorTests)

- **1**

- `int i = 0, j = 1, k = 15;`
- `k = i+++j;`
- `System.out.println(i + " " + j );`
- `System.out.println("k = " + k);`

- **2**

- `int myInt = 128;`
- `byte myByte;`
- `myByte = (byte) myInt;`
- `System.out.println(myByte);`



Reading user data & outputting admin info

# CONSOLE INPUT / OUTPUT



# Console input - old style

- `class ReadConsole`
- `System.in.read( );`
  - will wait for a character
- `int c = System.in.read( );`
  - will wait for a character and store it as an int
- `char c = (char) System.in.read( );`
  - will wait for a character and store it as a char



# Console input - better

- `Scanner inputStream = new Scanner(System.in);`
- `String inputLine = "";`
- `System.out.print("Enter your string: ");`
- `while ((inputLine = inputStream.nextLine()).length() > 0) {`
- `System.out.println("You said: " + inputLine);`
- `System.out.print("Keep typing to continue... (<Enter> to quit)");`

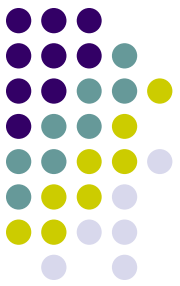




# Consoles

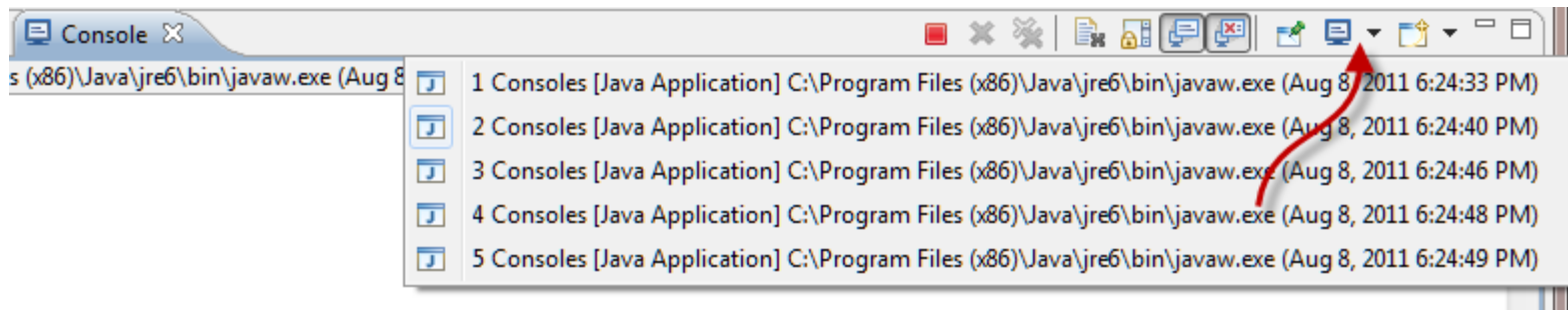
- Red button on console menu lit up means program is executing
  - Abort program by clicking red button
- **Eclipse** - Type on the console to input data.



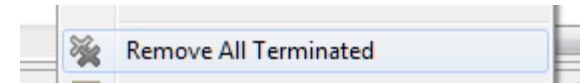


# Console management

- Multiple consoles can hide so look for enabled console button
- Drop down will select active console.



- After killing an app, remove it with right-click, Remove All Terminated.





# Console output streams

- `System.out.println("text");`
  - will print out and go to next line
- `System.out.print("text");`
  - will print out and wait at end of line
- `System.err.println("text");`
  - will print out to error stream
  - directed to console for Windows
  - Eclipse will use red text



# Eclipse terminal

- Open a Terminal (Window)
- Select type of terminal in Terminal window (local)
- Navigate to workspace/project/bin directory
- Execute: `java packageName.ClassName`

# Running from the command line

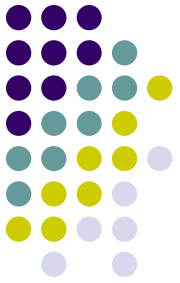


- Navigate in Windows to the package that holds the byte code (classes in bin folder)
- Shift + r-click the directory holding the package
- Select Open command window/ PowerShell here
- Type: `java <packageName>.<ClassName>`  
command line arguments



# O/S stream redirection

- `java wednesday.program 2> error.log`
  - Sends error output to file



Static methods and variables

# METHODS



# Definition

- A method is a function, a procedure, a subroutine, or a named unit of code.
- It can take multiple values of input to work on
- It can output one value
- It must be nested in a class



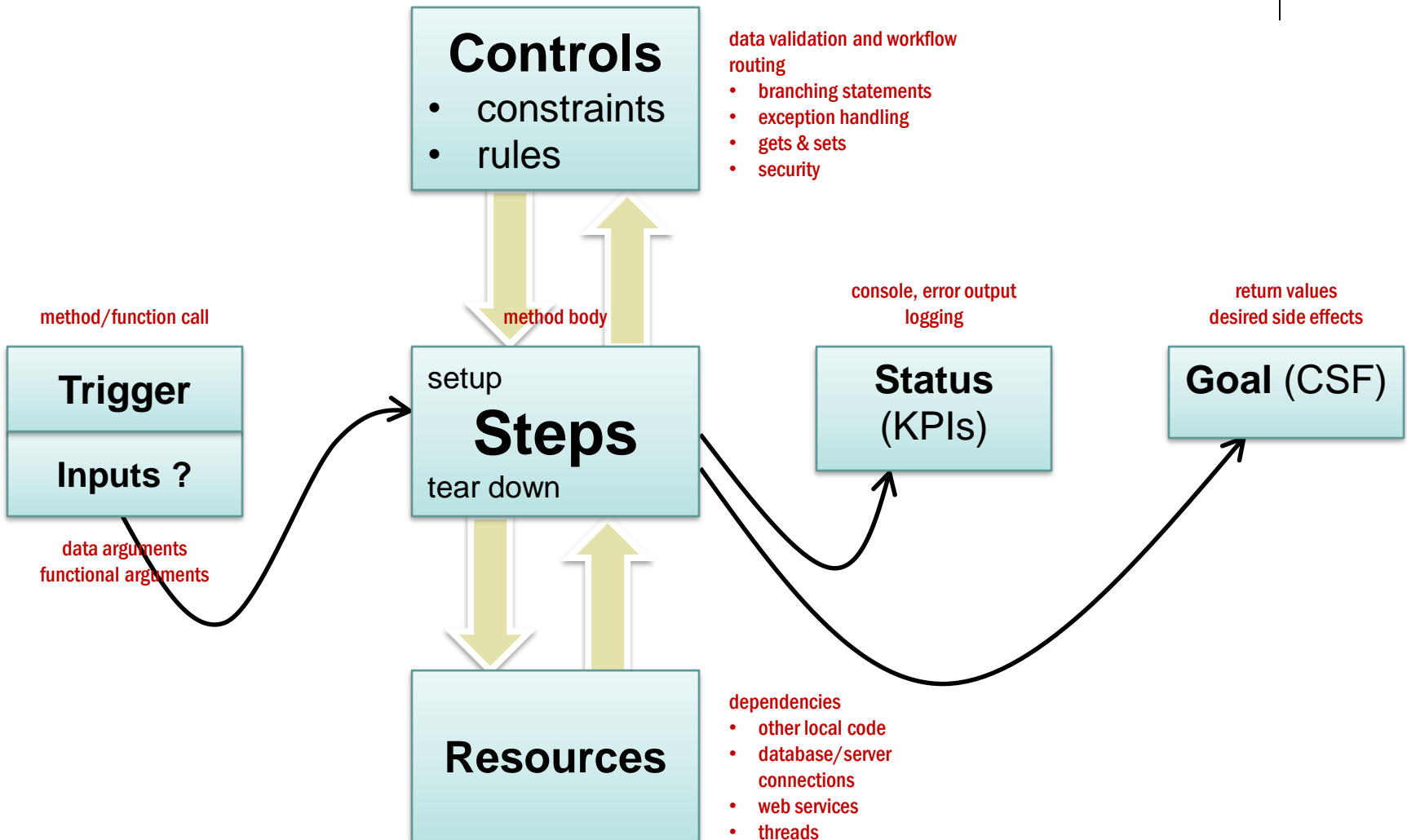
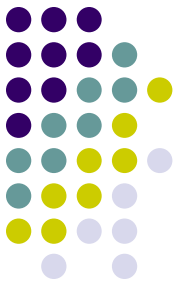
# Business requirements to code



- Business processes equate to methods/functions.
- The best designed methods follow requirements based on them in
  - clarity - naming and readability
  - sequence - steps in the workflow
  - reusability (low coupling / high cohesion)
  - uniqueness of effect (idempotency)

# A process model

the process parts in computer language





# Method syntax

- access modifier (public, private)
- static (optional)
- return type (output datatype)
- method name
- parameters – in parentheses (input datatype and local variable identifier)
- exception handling syntax
- method body – in curly braces



# Access modifiers

- Programmatic restriction based on where you are calling the method
- Must have physical access first
- **Private** – only accessed in the class
- none – only accessed in the package
- **Protected** – only accessed in the package and by classes with inheritance relationship
- **Public** – no restrictions



# Access modifiers

- Methods should be **public** as a rule
  - `public void printForm( )`
- Any public method is part of the class interface (the exposed behavior)
  - javadocs will summarize
- Methods can be **private** so that no one outside of its nesting class can access it.



# Static methods

- Called directly from the class they are nested in
- Procedural in nature.
  - Do not use objects
- These methods must use the word **static**
  - **public static void printForm( ) { ... }**



# main

- The main method is called by the JVM when starting an application
  - **public static void main**(String[ ] args ) {...}
  - args is a traditional name but not required
- Use for unit testing of class methods
  - Improve unit testing with junit or other frameworks
- One class will contain the “real” main method of the project.



# Exercise

- Write a main method that calls other main methods from classes in all of the packages that you have created so far.
  - Use the value of null for the `String[ ]` args argument.





# Method naming

- Follows identifier rules
- Use a strong verb to describe
  - calculateTax( )
  - printForm( )



# Method output

- No output requires the keyword **void**
  - `void printForm( )`
- Any output requires
  - the return type before the method name
  - a **return** statement in the method code block that returns data matching the return type
- The call to this method becomes this value
- `public double calcTax(double rate, double price)`  
`{`
  - `return rate * price; }`



# Method input

- If input is needed, a temporary variable name and the acceptable data type are used (a parameter)
  - `calcTax(double taxRate)`
- Multiple parameters can be used
  - `calcTax(double taxRate, String state)`



# Process – Test driven development

- Write the method call so it's readable and says what you want to do.
  - Run – it **fails**
- Add the method stub (nothing in the code block except a return if necessary)
  - Run – it **passes**
- Implement the method
  - Run – it **passes**



# Let Eclipse write the method

- Write the call to a method that doesn't exist
- Click the lightbulb on the left side  
or  
hover on method and click link
- Choose the **Create Method** option
- Move the method to where you want.



# Method signature

- The way that the JVM matches up the execution call to the right method.
- Includes
  - Name
  - Any parameters (match order and type)
- Does not include
  - Access modifier
  - Any output



# Overloading methods

- Overloading is using the same name but different parameters (order or type)
  - `calcTax( )`
  - `calcTax(int ratePercent)`
  - `calcTax(double rate)`
  - `calcTax(double rate, String state)`
  - `calcTax(String state, double rate)`
- This allows flexibility
  - `System.out.println( )`, `System.out.println(6)`, etc.



# Method code block

- The code block immediately follows the method name and parentheses
- The code block is executed when the method name is called
- The implementation





# Calling a method

- In order to call (execute) a method you must
  - invoke it with the right method name
  - invoke it with the right inputs (arguments)
  - have physical & logical access to the method
    - reference to code in project
    - **import** that code in class
- A static call uses the enclosing class name
  - `Reports.printForm( );`
  - `Revenue.calcTax( );`



# Calling a method - shortcuts

- A full call to a method with package name
  - `animals.Dog.printSpecies( );`
  - `monday.HelloWorld.main(null);`
- If you call within the scope of the enclosing package
  - `Dog.printSpecies( );`
- If you call within the scope of the enclosing class
  - `printSpecies( );`



# After the call

- A void method is only called
  - `Reports.printForm( );`
- A method call with output becomes a value.
  - `double tax = Revenue.calcTax( );`
- The value can be an input to another method
  - `ShoppingCart.add(Inventory.find("SZA123"));`



# Math

- Math contains many static math functions
  - `Math.sqrt(4)`
  - `Math.random()`
  - `Math.pow(2,8)`
  - `Math.round(2.4)`
  - `Math.abs(-2.7)`
- Math contains static values
  - `Math.E`
  - `Math.PI`



# Exercise (MathTests)

- Call in a main method
- Calculate the area of a circle ( $\pi * r^2$ )
- Calculate the volume of a sphere ( $\frac{4}{3} * \pi * r^3$ )
- initialize variables
- call methods
  - `double result = calcAreaOfCircle(radius);`
  - `// print`
  - `result = calcVolumeOfSphere(radius);`
  - `// print`



# Exercise (StaticMethods)

- Write and call a static method that
  - has no inputs and no outputs
  - has a String input and no output
  - has no input and a String output
  - has a String input and a String output
  - has two String inputs and a String output
- print return results
- print parameter values in method
- print status message in method (debug)



# Exercise – exercise headings

- Create a class(Exercise) with a method (printGroupWithHeading( )) that takes one String and calls a group of exercises. Before each exercise is output, print a formatted heading that uses the string in the heading. Use another class (PrintExercises) with a main method to start the printing process.