Fundamentals of Java

Why Java?

- Java provides platform independence with the same compiled binary running on any system that supports Java.
- The Java Virtual Machine (JVM) can interpret the binary code. This is more efficient than source-text interpreters, but still slower than native binary.
- The Just In Time (JIT) compiler can convert the Java binary (or “bytecode”) into native binary during the execution of the program. This allows finely tuned platform-specific optimizations not readily achieved with a traditional compiler. After this conversion, the code runs at similar speeds to pure platform binary programs.
- Java supports the popular object-oriented programming model, which has been demonstrated to facilitate maintainability of programs (if used properly!)
- Java also provides some support for the re-emerging functional programming model which is becoming increasingly relevant for concurrent code.

Java “Hello World” and Introduction to the IDE

- Create a HelloWorld project in your IDE
- Create a class, give it a package and a main entry point:

```
public static void main(String [] args)
{
  /* your code here */
}
```

- A public class ***must*** be defined in a source file that has a name that exactly matches the class name, in a directory structure that exactly matches the package structure, so the class HelloWorld, in the package, com.hello, must be located in a file .../com/hello/HelloWorld.java
- Package names should be all lower-case to allow for case insensitive file systems.
- Generate output with
  `System.out.println(data);`
- In Java, everything is defined inside a class (or other type), though a program is typically composed of many classes.
- Read from command line using the class: `Scanner`
  ```
  Scanner sc = new Scanner(System.in);
  String s = sc.nextLine();
  ```
  - Scanner can also read properly formed numbers using `sc.nextInt()`, `sc.nextLong()`, `sc.nextFloat()`, and `sc.nextDouble()`
- Library classes, such as Scanner, typically need import statements which immediately follow the package statement:
  `import java.util.Scanner;`
- IDEs usually offer a shortcut to generate import statements.

Lab:

1. Create and run a very simple example that prints a message of your choosing.
2. Read a name from the input and modify the output message using that.

Java variables, types, assignment, expressions, and operators

- Primitive types: `int`, `long`
    - Literal forms for decimal, hex, octal, and binary: `1_234`, `0xCAFE`, `077`, `0b10101`
    - Use (optional) underscore between digits where a comma might help the readability of a number
    - L suffix indicates a long literal
- Floating point: float, double, default literal is double
    - F suffix
- Java variables must generally be declared with their type, and must be definitely initialized before first use. E.g:
  ```
  int count = 0;
  ```
- The var pseudo-type can be used in a declaration provide that declaration is initialized. In this case the type is inferred from the initializing value, e.g.:
  ```
  var name = "Albert"; // name is a String
  ```
- Java has static type safety. Within limits cast operations can change the type of an expression:
  ```
  long x = 99; int i = (int)x;
  ```
- Arithmetic promotions readily change the type of numeric expressions to “wider” forms, but not narrower. All arithmetic operators produce results that have the type of the larger data type in the operation, but at least `int`.
- Identifiers should start with a letter and a sequence of letters, numbers, and perhaps underscore, follow. Java’s character set is unicode, so international characters are readily supported.

- Java has strong naming and stylistic conventions. Variables and methods should start with a lowercase letter, classes start with an uppercase letter.
- Choose clear, meaningful names, don't be afraid of long names. Use the IDE to "refactor" names if you realize they are misleading.
- `boolean` type: only boolean and Boolean are suitable for use in tests; Java does not have a notion of "truthy/falsy" . Literal values are `true`, `false`
- Overview of operators
    - Arithmetic: `+ - * / %`
    - Comparison: `< <= >= > == !=`
    - Logical: `&& || !`
    - Conditional: `? :`

Lab: Zeller's Congruence 1, Temperature Converter 1

Primitives and references

- 8 and 16 bit signed numbers might be useful in large arrays. These are supported by the primitive types `byte` and `short`
- Single characters are represented by `char`. This is unsigned but supports arithmetic. The literal form uses single (straight) quotes and supports unicode literals and escapes (which can also be used in String literals). E.g.:
  `char ch = 'A'; char xi = '\u559c';`
- An `enum` (enumerated type) is a special kind of class that allows representation of a fixed set of programmer-defined values. Take care to use if only for situations where the number of values required will not change during the life of the program. It will be necessary to edit, recompile, and redeploy the program

to make such a change:

```
public enum Suit {
  HEARTS, DIAMONDS, SPADES, CLUBS;
}
```

- Classes (and enums and other Java top-level features not yet introduced) in other packages can be referred to by their “fully qualified” names, that is, the name including the package string in dotted form. More normally, however, an import statement is used to make it possible to refer to them by just the basic name without the package.
- The class `String` represents character sequences.
    - Strings are immutable, changes require creating a new string.
    - The plus operator concatenates Strings. Adding a string to something else converts the other thing (anything!) to string.
- `==` verifies if the value of two expressions is the same. It has the expected effect with primitive types. But ***any*** other type is a pointer, or reference, to an object. So, `==` typically says “are these two the same block of memory?”
- The `equals` method is intended to encapsulate “do these represent the same thing”, but might not always work depending on the type it is used with. Use the equals method to compare `String` contents.
- Equality is a complex concept with aggregate types; take great care with it.
- Java documentation is available via `http://oracle.com`

    - `String` operations: `charAt`, `length`, `substring`, `toUpperCase`, `toLowerCase`, `trim`, `toString`
- The class `StringBuilder` represents mutable character sequences
    - Operations: `append`, `charAt`, `delete`, `deleteCharAt`, `replace`, `setCharAt`, `substring`, `toString`
    - The `equals` comparison doesn't work with `StringBuilder`
    - Construction of a `StringBuilder` from a `String` is supported.

Lab: Temperature Converter 2, String Comparisons, String Chewing, That's Mister To You!

Conditional constructs, if/else, and switch/case

- `if`/`else`: requires a boolean expression for the test:
```
if (myString.length() > 3) {
    System.out.println("That's long");
} else {
    System.out.println("That's short");
}
```
- `switch`/`case`/`default` performs selections based on exact matches with constant expressions (if can be used with variables, case cannot). The expression type must be one of `int`, an `enum`, or a `String`
- Since Java 17, a newer syntax for switch is provided, one change is that alternatives can be listed using commas, rather than multiple use of case.

- Another change at Java 17, which should be preferred where available is the "arrow" form. This form is less prone to errors:

```
switch (x) {
  case 98, 99 ->
    out.println("almost 100");
  case 100 -> out.println("it's 100");
  case 101, 102 ->
    out.println("a bit over 100");
  default ->
    out.println("Something else");
}
```

- Also, since Java 17, If used in a position that expects an expression, a switch becomes an expression and must provide a value or exception for all possible input cases. This can be done with either arrow, or colon forms, though the arrow should still be preferred:

```
int age = 10;
String message = switch (age) {
  case 8, 9, 10, 11, 12 -> "Hello, I
guess you're a 'tween'!";
  case 13, 14, 15, 16, 17, 18, 19 -> {
    String greeting = "Hi";
    yield greeting + " teenager";
  }
  default -> "Hello!";
};
```

  Note the use of the word "yield" which is required if this form is used with colons, or with multiple statements to the right of an arrow. Also, any time multiple statements are used to the right of an arrow, they must be grouped using a block.

- Since Java 21, further additional syntax for switch is provided, allowing a switch to make a choice based on the type of the target, and execute additional tests. In this usage switch can test arbitrary objects, rather than the restricted types previously allowed, and null can be tested to:

  ```
  Object obj = "Hello";
  System.out.println(switch (obj) {
    case String s when s.length() > 3 ->
       obj + " Java 21 world!";
    default ->
       throw new RuntimeException(
            "That can't happen!");
  });
  ```

Lab: Zeller's Congruence 2

Basic Loops

- Loops are controlled by `boolean` expressions
- The `while` loop might not execute at all:

  ```
  while (/* some test */) {
    // repeat this block
  }
  ```

- The `do`/`while` loop is similar but definitely executes at least once, because the test is made after the body. Note also, the required terminating semicolon:

  ```
  do {
    // repeat this block
  } while (/* some test */);
  ```

- Many loops take a three part form: initialize some variable(s), repeat the loop while some condition is true, do something before the next test / iteration. The original form of for loop provides directly for these three parts and is well suited to simple counting loops:

```
for (int x = 0; x < 10; x++) {
  System.out.println("x is " + x);
}
```

- Premature exit from a loop can be arranged relatively cleanly using the `break` or `continue` statements, which can accept labels to refer to an outer loop:

```
for (int x = 0; x < 10; x++) {
  if (x == 3) continue; // skip 3!
  System.out.println("x is " + x);
}

for (int x = 0; x < 10; x++) {
  if (x == 3) break; // print 0,1,2 only
  System.out.println("x is " + x);
}
```

Lab: Zeller's Congruence 3, Las Vegas 1, In Range 1, In Range 2, Text Alignment 1, Text Alignment 2, Wake Up

`List` and `Map`; elements of generalization

- Programs often need to store “many of something”
- The `List` interface and the `ArrayList` class provide for this (among other library features)
  - The capabilities of a `List` include inserting, looking at, fetching, asking how many, managing the order of items
  - A `List` allows duplication of items in the contents
  - Finding things requires properly implemented `equals` behavior, which can be a problem in the general case but works reliably with `String` objects.

- The type of thing in the `List` can, and generally should, be qualified. The right hand side of this example illustrates the so-called "diamond" operator, which allows the type of the value being created to be inferred from the context to which it is being assigned.:
  ```
  // List restricted to use with String
  List<String> ls = new ArrayList<>();
  ```
- A `Map` is used to store key-value pairs, facilitating fast lookup. This is typically implemented using the `HashMap` class
  - This provides a table, similar to a trivial database with a primary key and single additional column
  - Look items up by key
  - Change the value associated with a key
  - Delete a row
  ```
  Map<String, String> table =
      new HashMap<>();
  // insert / update
  table.put("Boss", "Fred Jones");
  // read / select
  String boss = table.get("Boss");
  // delete
  table.remove("Boss");
  ```
- Notice that `List` and `Map` are examples of an important Java syntax feature called an interface. An interface represents a “generalization” and generalizations allow us to discuss “what something can do” and “how to use something” without any reference to “how something achieves its results”. This allows us to change the specific implementation, much as we might drive a different car without having to

relearn how to do so. Key elements of the "interface" of a car would be the pedals, and the steering wheel–so long as these have the same effect, we don't need to concern ourselves with the exact make and model of the car. This is a central and frequently-used aspect of object oriented design and greatly facilitates maintenance.

- `List` and `Map` cannot handle primitives directly, but use “wrapper objects” instead. So, an `int` can be stored as an `Integer`. Generally, the 8 primitives are represented by objects that have the same name but a capital first letter, the slight oddity is that while `int` and `char` are abbreviated, they become `Integer` and `Character` in the wrapper forms. Generally, direct assignment between primitives and these wrapper types is supported, and the conversion happens automatically. This is referred to as "autoboxing" and "auto-unboxing".

Lab: Students, Favorite Foods

The “enhanced for” loop

- The “enhanced for” loop can be used to process every element in a Collection in turn. A `List` is a kind of Collection, and many other types exist.

```
List<String> names = ...
for (String name : names) {
     System.out.println("Name " + name);
}
```

Lab: Students 2, Favorite Foods 2, Las Vegas 2, 3a, and 3b, Code Breaker, Guessing Game

Arrays

- Java provides arrays of any type, though Lists or other collections are often preferred
- Initialized literal form:
  ```
  String [] names = { "Fred", "Jim" };
  ```
- Arrays do not have to be initialized literally:
  ```
  String [] tenNames = new String[10];
  ```
- Read/write an item in an array by index number:
  ```
  names[0] = "Frederick";
  ```
- Arrays are fixed size after creation, and can report their size:
  ```
  System.out.println("There are "
       + names.length + " names");
  ```
- Arrays can be of arrays, allowing multi-dimensional effects, in both rectangular and non-rectangular forms.
- Arrays can be used in the enhanced for loop.

Static methods; Java’s “subroutines”

- Operations that are frequently used, or which form a “meaningful chunk of behavior” can (and should) be separated out into a well-named, standalone operation called a “method”.
- Data is passed into a method using arguments which get a copy of the value in a local variable called the formal parameter.
- Methods can return a result to the caller. The type of the returned value must be predefined, and the value must be returned. If no return is wanted, declare a `void` return. This kind of method is sometimes called a "procedure", and might be less good style in modern code.

- Example method declarations:

```
// accept two values and return the sum
public static int addUp(int a, int b) {
     return a + b;
}
// accept a single name String,
// greet that person, return nothing
public static void greet(String name) {
     System.out.println("Hi " + name);
}
```

- Call the method using the name, and argument list (which will be copied to the formal parameters) in parentheses:

```
greet("Jim");
```

- If the method declares a non-void return type, the call is an expression of the returned type and can be used anywhere an expression of that type would be used:

```
int sum = addUp(3, 9);
sum = addUp(sum, 6) + 4; // sum is 22
```

- The type and order of formal parameters is part of the identity of the method, so two methods can have the same base name:

```
public static int addUp(int a, int b) {
return a + b; }
public static int addUp(int a, int b,
int c) {
     return a + b + c;
}
```

  This is called “method overloading”. Which method to call is determined by the compiler, based on the arguments in the ***call***.

- A method can take a variable length argument list for the last element of the formal parameter list. If declared using the ellipsis format shown here, then the invocation can simply be a comma separated list of items of the appropriate type and the compiler will create and pass an array as the single "real" argument.
```
public static void showMany(
  String ... strings) {
  for (String s : strings) {
    System.out.println("> " + s);
  }
}
// invoke using:
showMany("Fred", "Jim", "Sheila");
```

Lab: Zeller’s Congruence 4, Calendar, Palindrome Checker

When things go wrong, using exceptions

- An exception is an “out of band” mechanism that reports that something broke.
- The exception is an object, but is not passed as a result through the normal return channel of a method.
- The type of the exception represents the category of failure, and it should contain a message that is more fully descriptive.
- An exception may be thrown by any code, custom-written, library, and even some operators. Special exceptions can be custom-created to represent specific problems.
- To represent a problem, throw an exception:
```
if (databaseBroke) {
      throw new SQLException("Broken!");
  }
```

- If code might throw an exception, one of two techniques is appropriate; attempt to handle the problem, or report the problem to the caller.
- To handle the problem, use a `try` / `catch` structure:

```
try {
  String result = readDatabase();
  System.out.println(
    "Result is " + result);
} catch (SQLException ex) {
  System.out.println(
    "Problem " + ex.getMessage();
}
```

- To report the problem to the caller of this method:

```
public static void dodgy()
    throws Exception {
  if (somethingBroke) {
    throw new IOException("Ooops");
  }
}
```

- Some exceptions, called "checked exceptions" mandate that they are either handled with a catch or declared on the method that fails to handle them. Others, subtypes of `RuntimeException` or `Error` represent problems that should be resolved by fixing the code or the design, rather than retrying. These types do not mandate any code to address them.

Lab: Divide By Zero, Say Please!

Working with multiple classes

- Any code in Java must be part of a class (or similar construct). A class is--from a certain perspective, a kind of (very small) "module", therefore many classes are typically needed to build real software. In this sense, the term "module" simply means a collection of closely related code elements that address one concept. This module concept refers to a simple grouping has a major purpose of making it easier to know where to look in a large codebase when trying to make changes. It's important to not that since Java 9, there is a specific language feature called "module" which is a very much more specific thing. Conceptually, a Java module (sometimes called a JPMS-Java Platform Module System–module addresses this same problem, but at a higher level of abstraction.)
- Classes belong to packages, which create a kind of namespace. Like members of a family, a class can refer to another in the same package by its simple name. Packages also provide for the grouping of related code items at a level higher than classes, but lower than actual JPMS modules.
- In the absence of any other action, classes in other packages must be referred to by their fully qualified name (like using the full name when referring to someone in a public space). However, this can be circumvented by using an `import` statement, which arranges for the class to be addressable by the short name (rather like inviting a friend round to stay with the family).

Essentials of “classpath”

- Classes must be located in directory structures that map exactly to their package names (and be in files that match the classname)
- Multiple such directory structures can be combined for a single program
- The roots of these directory structures are referred to as the classpath. The classpath is searched from left to right to find each necessary class.
- Classpath is usually specified using -cp on the execution command line. Other means exist.
- Classpath can include directories, and also archive files in either .jar or .zip formats (they’re essentially the same format).
- Java’s core libraries are special, they live on the bootclasspath, and are not normally altered.
- Third party libraries are handled by adding the archive file to the classpath.
- Managing the library requirements of a program during compilation and execution can be complex, and is often handled by special software. “Maven” is a popular example.
- Under the JPMS system (beyond the scope of this course) this concept is modified to a "module path", which performs a conceptually similar role to that of classpath.

Optional Lab: Under your instructor’s guidance, create two classes, such that one acts as a library, and the other makes use of the library. Place the classes in entirely different trees, then compile and execute them from the command line.

Structured or aggregate data

- Most real programs need to handle structured data, such as a representation of a “Customer” with name, address, credit limit, and perhaps many other attributes.
- Java allows such structures to be defined using classes (though classes are typically much “more” than just this).
- To define a class for this purpose the following syntax serves:

```
public class Customer {
  public String name;
  public String address;
  public long creditLimit;
}
```

- Given this, a variable may be declared and initialized:

```
Customer joeThePlumber = new Customer();
```

- ***IMPORTANT NOTE***: The variable `joeThePlumber` is a “reference” (or pointer) to the structured data in memory. Simply declaring the variable does ***not*** create storage!
    - So, this fails:

      ```
      Customer albertTheCarpenter;
      albertTheCarpenter.name = “Albert”;
      ```

      This typically causes a `NullPointerException` though it might also fail to compile, depending on the location where `albertTheCarpenter` is declared.
- The various fields are accessed using a dot notation:

```
joeThePlumber.name = “Joe”;
joeThePlumber.address = “123 Any St.,
```

```
Smallville";
joeThePlumber.creditLimit = 20_000;
```

- Dotted access is typically read/write:

  ```
  if (joeThePlumber.creditLimit > amount)
  {
    // Joe can buy the goods
  }
  ```

- Notice that the ***class***, in these examples is a "template" for "how to make a Customer". When we say ***new Customer()*** we actually create an ***object*** of the type built according to the specification in the template.

Lab: Birthdays 1

From structures to objects

- The usual approach of object oriented design is to tightly associate code that works on a data structure with the structure itself.
- Only the privileged code is able to modify the data, and if the data elements are inconsistent we know where to look for the root problem.
- Operations on the data are all described by methods, we don't have to understand how the representation works, only what behaviors we can apply. This is closely parallel to a floating point number representation which supports operations like add, subtract, etc.
- The syntax that restricts data access is to mark the variables `private` instead of `public`:

  ```
  public class Customer {
    private String name;
  }
  ```

- To associate a behavior with the objects of this type, embed the behavior within the class, but omit the word `static`:

```
public class Customer {
  private String name;
  public String getName() {
    return this.name;
  }
}
```

- To invoke the behavior, use the dotted form:

```
System.out.println(
  joeThePlumber.getName()
);
```

- The prefix object in the call becomes `this` inside the method.
- OO sometimes refers to this kind of behavior invocation as “sending a message to the object”.

Labs: Birthdays 2

Instance and static features

- Instance features (fields or methods) are associated with, and accessed using a prefix of, a specific instance of the class. By contrast, static features have no particular association with any one object, but are related to the class as a whole.
- Static behavior has access to the private members of any object of the same class, but there is no `this` reference, that is, there is no “context” object, so any object that is to be manipulated by a static behavior must be passed as a parameter to that behavior.
- Static fields exist in the class definition, rather than one-per-object. Any object can access the field, but it

does not have its own value. So, for example, a `FordFocus` class might have a `speed` instance field, since every one of these cars can be moving at a different speed, but it also might have a static member `MAX_DESIGN_SPEED`, since the design top speed is a *single* value that relates to the design as a whole, not one-per-instance.

Lab: Cars

Encapsulation

- The keyword `private` can be applied to any element of an object in the class definition. This limits direct access to that feature to code within the enclosing top-level curly braces that surround the declaration.
- Applying `private` to the data items in the object allows a design to ensure that self-integrity rules (for example, the day of the month must be greater than zero, and less than the number of days in the current month of the current year.
- Designs should identify the “rules” for internal correctness of the objects they define, and ensure that they’re implemented, and verified, consistently.
- Encapsulation requires that a newly minted object must have valid fields. For this we can use the so-called constructor.
- A constructor is a behavior that is used to initialize an object immediately after the memory is allocated and before the reference is given to the caller. It looks like this:

```
public class Customer {
  private String name;
  // constructor follows
```

```
    public Customer(String name) {
      this.name = name;
    }
  }
```

- A constructor:
    - Has exactly the same name as the class.
    - Can take arguments, or none, as required
    - Does ***not*** declare a return type (the “return” is implicitly the object being initialized).
    - Cannot be labeled `static` (it ***must*** initialize an instance)
    - Can be public or private as needed.
    - Must (generally) exist with the right parameter type list if an object is to be created with those parameter types.
- Encapsulation can enforce the integrity of objects by rejecting attempts to create “bad” states. Throw an exception to achieve this.
- An exception can (and should) also be thrown from a constructor if the provided data are not suitable for creating a valid object.
- Since Java 16, a new language feature called a "record" provides a convenient way to define a class that has certain features provided automatically, but also enforces certain constraints that are intended to reduce the likelihood of programming errors. This code would declare a Customer with similar elements to those of the Customer described above:

```
record Customer(
  String name,
  String address,
  long creditLimit) {}
```

In this record, the fields are `private`, and must be set when the object is created using the automatically created constructor that takes the same form as the class declaration line. The fields are final, which means–approximately–that they cannot be changed after initial creation. Note that this is actually considered to be a design *advantage* because it's likely to reduce errors. The type is also capable of representing itself as text directly, and comparing instances for equivalence (it implements `toString`, `hashCode`, and `equals` methods)

Lab: Birthdays 3

Generalization

- An effective way to simplify maintenance is to minimize dependencies. One approach is to limit knowledge to “what something does” rather than “how something is done”.
- Encapsulation provides a means to do this, leaving only the behavior (method signature) visible.
- An interface is another way to limit knowledge. It declares what behaviors will be available, but hides the implementing class, so that class can be changed with no consequences to the code that uses the implementations.
- Declaring an interface is very similar to declaring a class, but using the keyword `interface`:

```
public interface Plumber {
  void mendPipes(Pipe toBeFixed);
  int estimateCost(List<Job> work);
}
```

- A Java interface defines behaviors for a type. It cannot define general storage (fields) and generally is restricted to declaring only the signatures of behaviors.
- Since Java 8 an interface is permitted to declare static methods, both private and public, private instance methods, and public "default" methods. A default method provides a "fallback" implementation that is used only if no suitable implementation is available in the class hierarchy at any level.
- Methods of an interface are implicitly public, and it's usual to omit the keyword. As noted, Java 8 and newer permits private static and instance methods, but nothing in an interface can have any accessibility other than `private` or `public`.
- An interface type can (and probably should whenever possible) be used to refer to an object of any class that "implements" that interface. Doing so reduces knowledge and dependencies, and reduces or eliminates the consequences of many changes.
- To create a class that implements an interface:

```
public class HumanPlumber
    implements Plumber {
  public void mendPipes(Pipe fixThis) {
    // Do human things to fix the pipe
  }
  public int estimateCost(List<Job> j) {
    return 500;
  }
}
```

- A class can implement multiple interfaces, simply list them as a comma separated sequence.

- A class that claims to implement an interface must (generally) provide implementations for all the methods of that interface. Exceptions to this rule are where the class inherits an implementation from a parent class or if a default method (Java 8 and newer) is provided.
- Importantly, when invoking a behavior on an object, the behavior that executes is the behavior of the *object*, not behavior associated with the type of the *variable* referring to the object (if the type of the variable is an interface, the variable doesn't even have behavior!) This is why the term "sending a message" is meaningful.

Lab: Things with Names

Object Oriented design essentials:

- Find "candidate" classes by looking at the requirements and finding nouns the represent significant parts of the problem domain. Things like "Customer", "Account", "Invoice" and similar. These things do not have to be physically real, they can equally well be conceptual (e.g. representing "business processes").
- Consider what generalizations might make multiples of these into a single concept.
- Consider what the essential nature (state and behavior) these things have.
- Consider how these elements are joined together into a working system.
- "Realize" (bring into reality) additional classes where needed to represent things like business process (a Customer does not process an invoice, nor does an invoice--do not get fooled into simply putting behavior into one of the already existing classes)

Guidance for optimizing a design

- Model “reality” so that anyone who understands the business domain, when reading the code, sees elements, and a structure, that seems “natural”.
- Use names carefully and accurately, these are the first thing a reader sees when trying to understand the code.
    - Modern IDEs make renaming things (and other “refactorings”) really easy. There’s no excuse for leaving something with a poor, unclear, or worst of all misleading, name.
- Ensure encapsulation, expose only behaviors (and constants)
- Minimize knowledge of other parts of the system:
    - Use interfaces not concrete classes
    - Minimize what client code needs to know about how the class “works”
- Minimize the consequences of change
- Keep together what belongs together, keep apart what belongs apart (“cohesion” and “single responsibility”)
- Keep apart what changes independently, keep together (in packages) what changes together

Key design patterns and related ideas in Object Oriented software development:

- Design patterns are conceptual templates that identify tried and proven ways to divide up responsibilities in the face of certain commonly recurring problems in software.

- Most design patterns typically either match based on “this part shouldn’t be affected by changes in that part” or “although logically part of that larger whole, this aspect changes independently of that, so let’s separate it out”.
- Having separated the parts, the pattern also describes how they should interact to do their job.
- It’s important to note that patterns are about maintenance, not about simple “functional correctness”. This is true of all “good” OO design, however.
- Many patterns are based on the observation that an object provides not only state (data), but also behavior (functions, computation, algorithm). Having a reference to an object means that we also have a reference to behavior. Changing the value of a reference variable can also change the behavior.

Pattern: The Strategy

- The strategy might be the most important pattern of all; it addresses the situation where *how* something is achieved can vary over the life of an object.
- Example: a bank account computes interest. Simply varying the percentage interest rate is trivial, and only requires a numeric variable. But changing the logic is harder. For example, if we need to be able to offer an introductory special, such that interest is given at a higher rate if the balance exceeds a certain amount for the first six months. This would then change later.
- The strategy can be implemented with these steps:
    - Defining an interface that specifies the operation (or perhaps operations) that changes

    - Defining as many classes as are needed to represent the variations on this behavior. Each implements the interface. These are actually the “strategy” objects.
    - Defining a variable in the class that represents the main concept being modeled
    - Defining the necessary internal behaviors so that they delegate to the strategy object through the variable.
    - Defining mechanisms to set and change the strategy variable as the needs of the main class change.
- It’s not always trivial to decide what parameters should be passed into the strategy object. Note that the parameters that first come to mind for the simple cases of behavior that first occur to us are often insufficient for later changes. Fortunately, the consequences of adding more parameters are not usually extensive.

Lab: Birthdays 4, Custom awards,

Pattern: The Command

- The command pattern separates some detail of how to perform an operation from the overall plan of action. The pattern expects that an operation is defined to accept an argument that is actually behavior to be used to fill in that detailed aspect for each invocation.
- Example: Sorting items in a list is mainly performed by comparing the relative ordering of pairs of items, and changing their order as necessary. However, the decision of what the relative order of two particular items should be can depend on many things. So, a sort mechanism can be constructed so that it allows the

ordering behavior to be specified at the time the sorting is requested, rather than being built in.

Lab: Birthdays 5

Pattern Group: Factories and Dependency Injection

- As a general rule, the user of a service should not be burdened with particular implementation knowledge about that service. We need “a plumber” not “Joe the plumber from down the street”. If we can avoid this kind of knowledge, it’s much easier to change the particular implementation (in the illustration, if Joe retires, or a robot becomes available that’s faster and cheaper).
- The factory concept says that instead of calling a constructor, we get access to utilities by calling a “factory” behavior that amounts to “get me something that can satisfy this need”.
- Instead of writing:

  ```
  Car c = new FordFiesta();
  ```

  We might write:

  ```
  Car c =
    CarRentalCompany.getEconomyCar();
  ```

- Dependency injection is a “framework” system that starts our software, and assigns values to otherwise uninitialized variables:

  ```
  @Inject
  Car c;
  ```

  In this case c would be initialized from “outside” (by the framework) based on a configuration file, or perhaps by some conventions.

Unit Testing

- Traditional software, in languages like C/C++, had to be compiled and linked into a large executable. In Java, and other more recent languages, every single class becomes like a “dynamically linked library”. This makes it possible to load a single class and test it in isolation.
- Testing in isolation makes it much easier to determine the location of the bug when a problem is found. This is called “unit testing” (since it tests individual units).
- If a unit requires other supporting code, these are replaced for the purpose of the test with a look-alike that is very simple, and only answers the questions necessary to make the test work. This look-alike might be called a fake, dummy, double, stub, proxy, or (perhaps most commonly) a mock.
- Unit testing can be very effective, but does not replace all forms of testing. In particular, end to end, acceptance, security, and performance testing are still necessary.
- Good unit tests are documentary; better yet, they form documentation that cannot be wrong (if they were wrong, the test would fail).
- A test should have the general form “given, when, then”. This means that it should set up a situation, trigger some behavior to be tested, and then specify what the result should be.
- Good tests, and the mocks that support them, should be simple (“so simple that they’re obviously correct”)

Introduction to classical inheritance

- Most object oriented languages provide a mechanism called “inheritance” (sometimes more precisely referred to as “implementation inheritance”).
- Inheritance conflates generalization with code reuse. Unfortunately, these two are not very often a good pairing. Hence inheritance should probably be used very sparingly in modern systems that provide generalization as a separate syntax mechanism.
- More unfortunately, the earliest OO languages only supported generalization through inheritance and so generations of OO programmers learned this as their “goto technique”. As a result, you’ll see it used a lot and should understand it.
- Inheritance describes a situation where generalization not only describes how something can be used, but also generalizes parts of the implementation.
- A “base class” (also called a parent class, or a superclass) is defined and includes both state and behavior (real implementations of that behavior, not just the specification of the method signature).
- Then another class (called a child class, a derived class, or a subclass) can be created such that it has everything in the parent without the code being copied. Additional elements can be added, and the existing behaviors can, if desired, be altered.
- To do this in Java, define a base class:

```
public class Animal {
  int legCount;
  public boolean hasHair() {
    return true;
  }
  public String getNoise() {
    return "undefined";
  }
}
```

And define a child class:

```
public class Lion extends Animal {
  @Override
  public String getNoise() {
    return "ROAR!!!!";
  }
}
```

- In this example, a Lion has a legCount, and a method hasHair (which returns true). These elements are "inherited". It also has a getNoise behavior, but if we ask a lion what noise it makes, it tells us it roars, rather than reporting undefined.
- The @Override annotation is not required syntax, and does not change the generated code or its behavior, however, it verifies that the method that follows is in fact a replacement of some inherited behavior. In this way, it can turn a simple spelling error that would create a bug into a compiler error that we can fix early.
- A class can both "extends" another and implement interfaces. In this case, the extends part comes first.
- Parent class features are typically available to the child using the `super` keyword in the same way that the `this` keyword might be used.
- Classical inheritance has some significant limitations:
  - It's a compile-time relationship. Unlike the strategy pattern, the behaviors that are inherited

can only be changed by recompilation, not by assignment.

- Each subclass gets one set of inherited, and one set of overridden behaviors. If several subclasses need various independently changing behaviors, these will have to be duplicated code. Again, the strategy pattern is preferable from a flexibility perspective, although the code required to implement the strategy can be clumsier.

Lab: Zookeeper