Fundamentals of the Java™ Programming Language

SL-110-SE6

Course Contents

Preface

About This Course

Course Goals

Upon completion of this course, you should be able to:

- Demonstrate knowledge of Java™ technology, the Java programming language, and the product life cycle
- Use various Java programming language constructs to create several Java technology applications
- Use decision and looping constructs and methods to dictate program flow
- Implement intermediate Java technology programming and object-oriented (OO) concepts in Java technology programs

Course Map

Introducing Java Technology Programming

- Explaining Java™ Technology
- Analyzing a Problem and Designing a Solution
- Developing and Testing a Java Technology Program

Explaining Java Technology Programming Fundamentals

- Declaring, Initializing, and Using Variables
- Creating and Using Objects

Dictating Program Flow

- Using Operators and Decision Constructs
- Using Loop Constructs
- Developing and Using Methods

Describing Intermediate Java Technology and OO Concepts

- Implementing Encapsulation and Constructors
- Creating and Using Arrays
- Implementing Inheritance

Topics Not Covered

- Advanced Java technology programming – Covered in SL-275: *Java™ Programming Language*
- Advanced OO analysis and design – Covered in OO-226: *Object-Oriented Application Analysis and Design for Java™ Technology (UML)*
- Applet programming or web page design

How Prepared Are You?

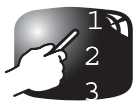
To be sure you are prepared to take this course, can you answer yes to the following questions?

- Can you create and edit text files using a text editor?
- Can you use a World Wide Web (WWW) browser?
- Can you solve logic problems?

Introductions

- Name
- Company affiliation
- Title, function, and job responsibility
- Experience related to topics presented in this course
- Reasons for enrolling in this course
- Expectations for this course

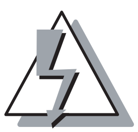
Icons



Demonstration

Discussion

Note



Caution - Electrical

Caution - Heat

Icons

Case Study

Self-Check

Typographical Conventions

- `Courier` is used for the names of commands, files, directories, programming code, programming constructs, and on-screen computer output.
- **`Courier bold`** is used for characters and numbers that you type, and for each line of programming code that is referenced in a textual description.
- *`Courier italics`* is used for variables and command-line placeholders that are replaced with a real name or value.

Typographical Conventions

- ***`Courier italics bold`*** is used to represent variables whose values are to be entered by the student as part of an activity.
- *Palatino italics* is used for book titles, new words or terms, or words that are emphasized.

Additional Conventions

Java programming language examples use the following additional conventions:

- `Courier` is used for the class names, methods, and keywords.
- Methods are not followed by parentheses unless a formal or actual parameter list is shown.
- Line breaks occur where there are separations, conjunctions, or white space in the code.
- If a command on the Solaris™ Operating System (Solaris OS) is different from the Microsoft Windows platform, both commands are shown.

Module 1

Explaining Java™ Technology

Objectives

- Describe key concepts of the Java programming language
- List the three Java technology product groups
- Summarize each of the seven stages in the product life cycle

Relevance

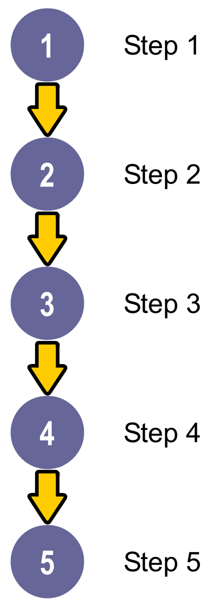
- What is the definition for the following words:
  - Secure
  - Object oriented
  - Independent
  - Dependent
  - Distributed
- What are the stages involved in building something, such as a house or piece of furniture?

Key Concepts of the Java Programming Language

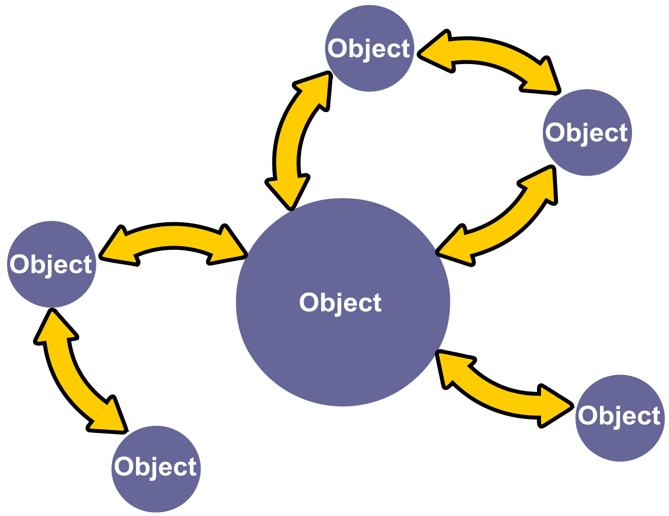
- Object-oriented
- Distributed
- Simple
- Multithreaded
- Secure
- Platform-independent

Procedural Programming

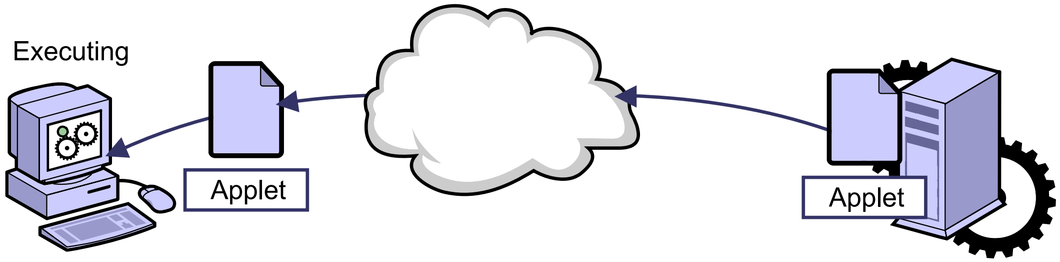
Procedural programming focuses on sequence.



Object-Oriented



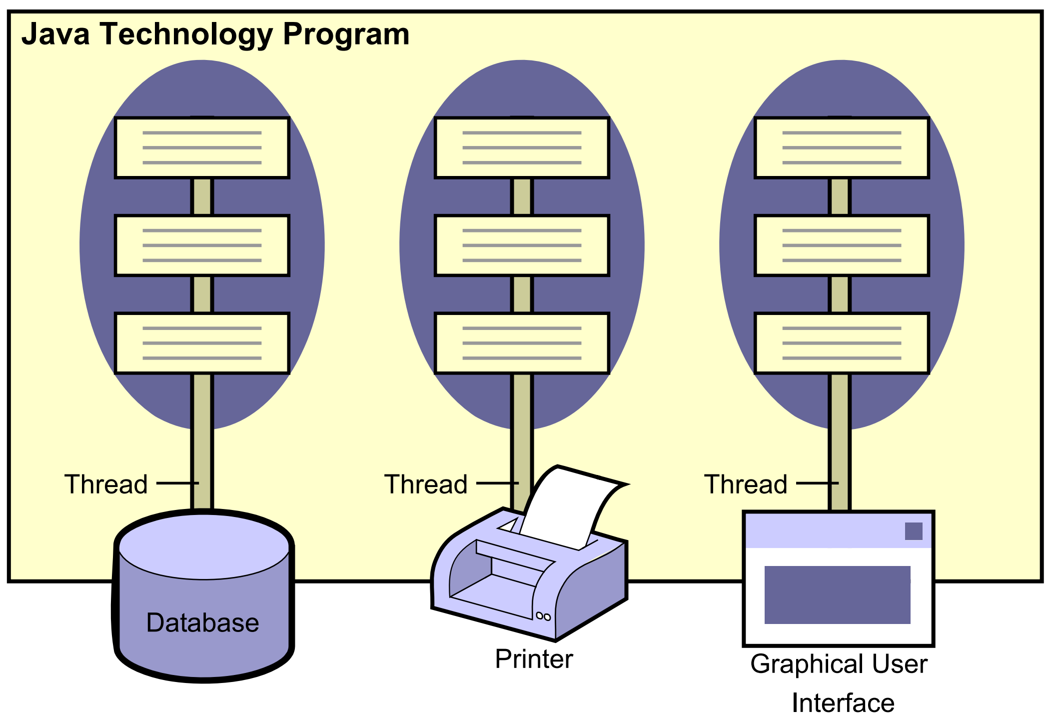
Distributed



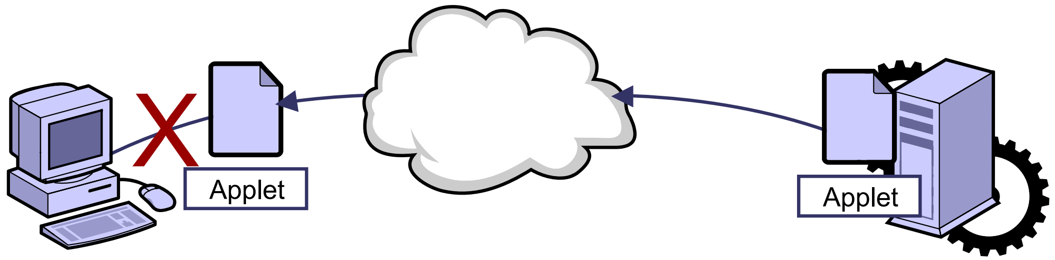
Simple

- References are used instead of memory pointers.
- A `boolean` data type can have a value of either `true` or `false`.
- Memory management is automatic.

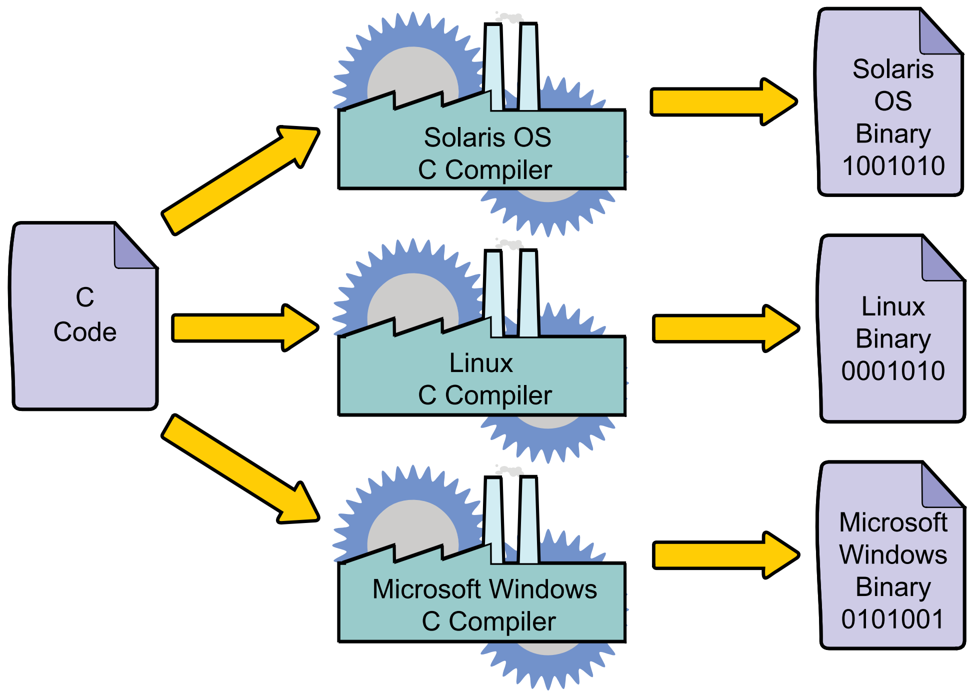
Multithreaded



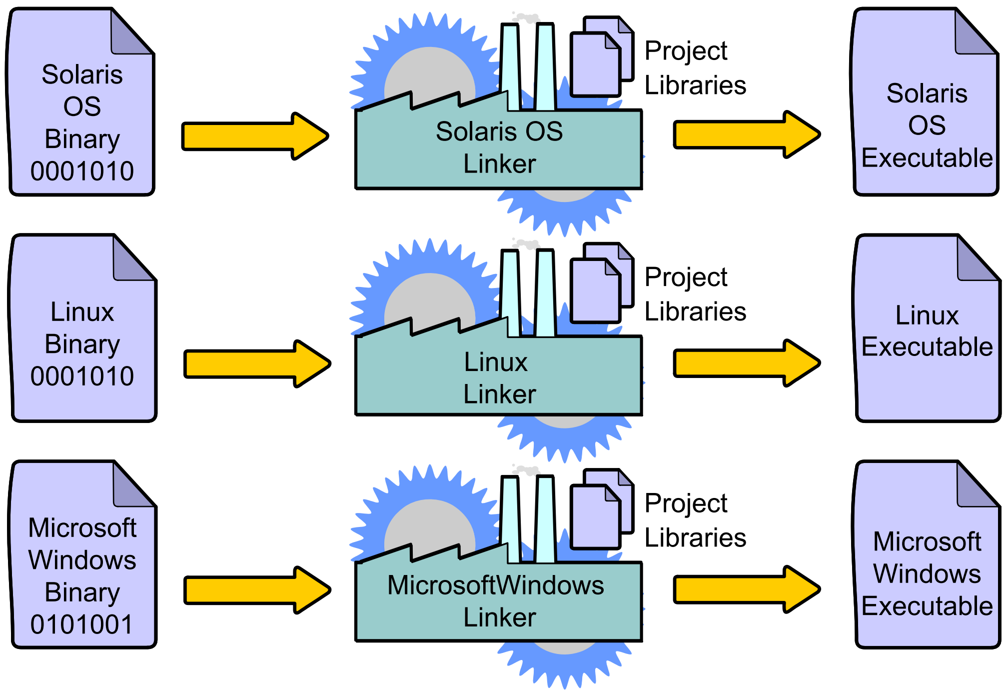
Secure



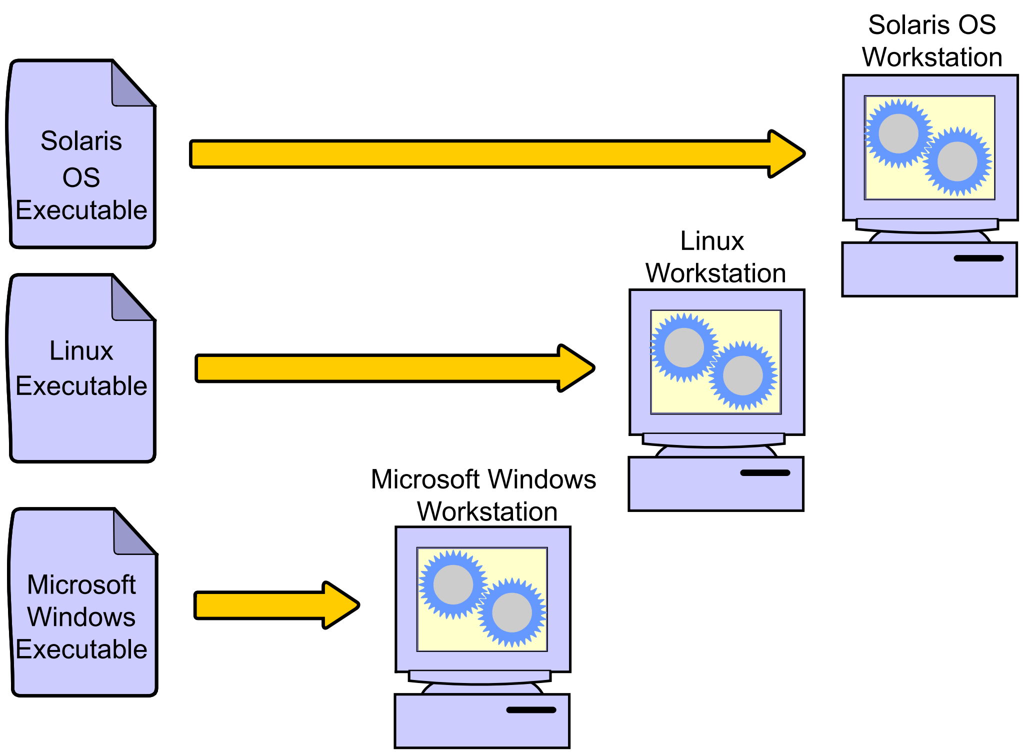
Platform-Dependent Programs



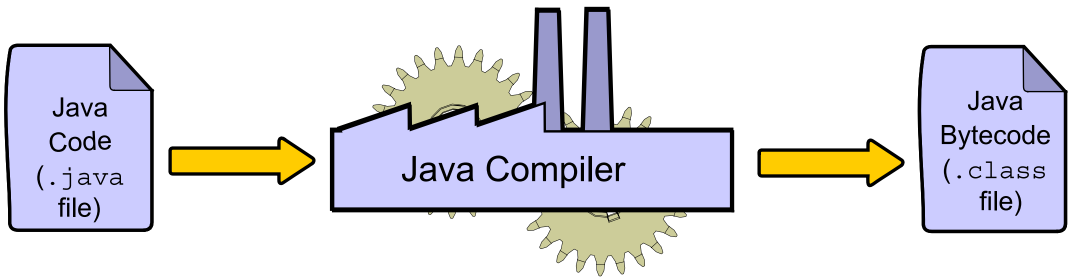
Platform-Dependent Programs



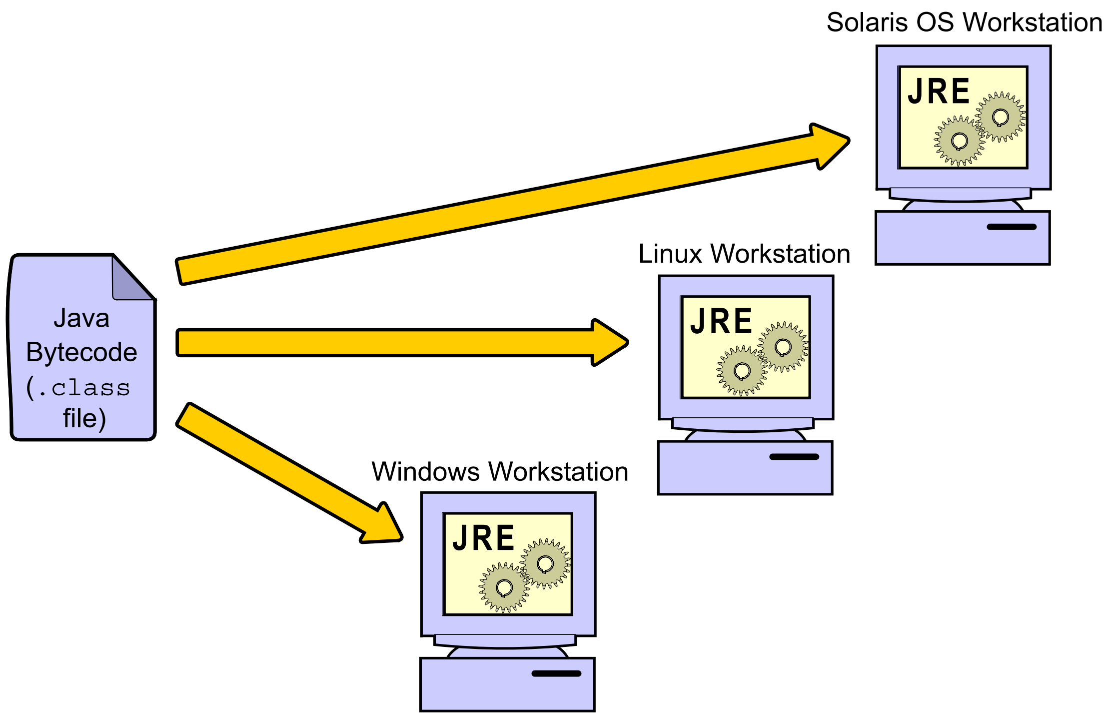
Platform-Dependent Programs



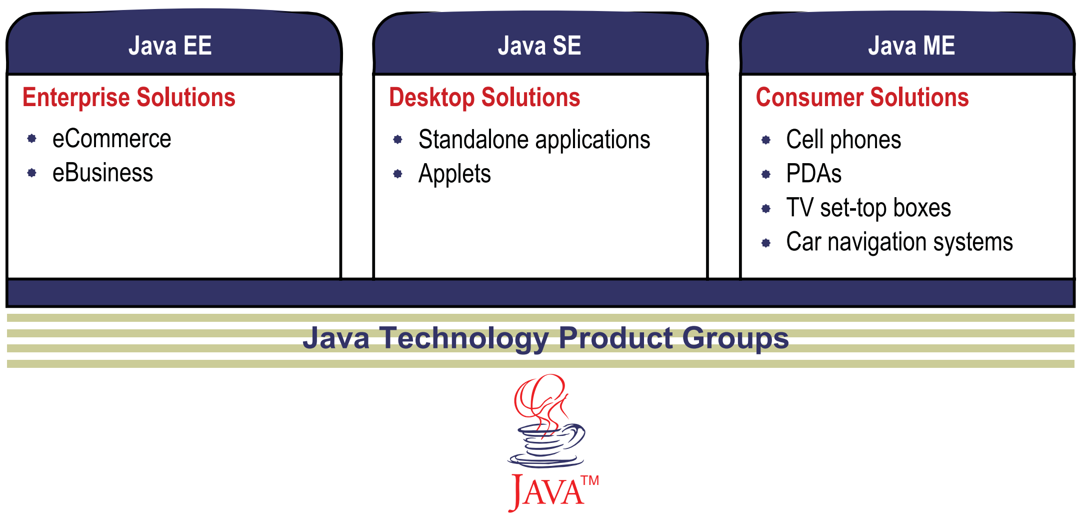
Platform-Independent Programs



Platform-Independent Programs



Identifying Java Technology Product Groups



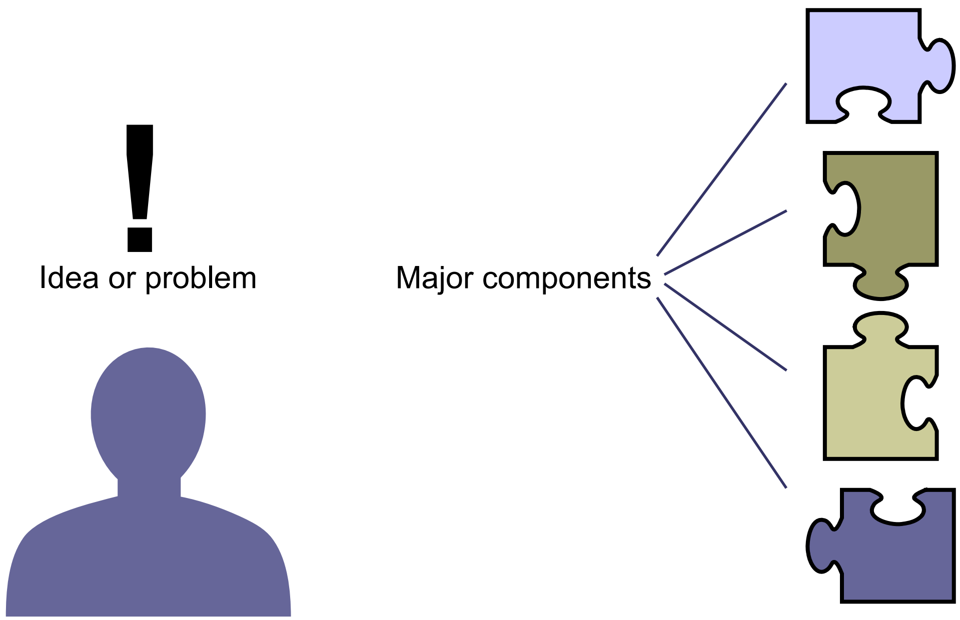
Using the Java Platform, Standard Edition SDK Components

- Java runtime environment (JRE):
  - A Java Virtual Machine (JVM™) for the platform you choose
  - Java class libraries for the platform you choose
- A Java technology compiler
- Java class library (API) documentation (as a separate download)
- Additional utilities, such as utilities for creating Java archive files (JAR files) and for debugging Java technology programs
- Examples of Java technology programs

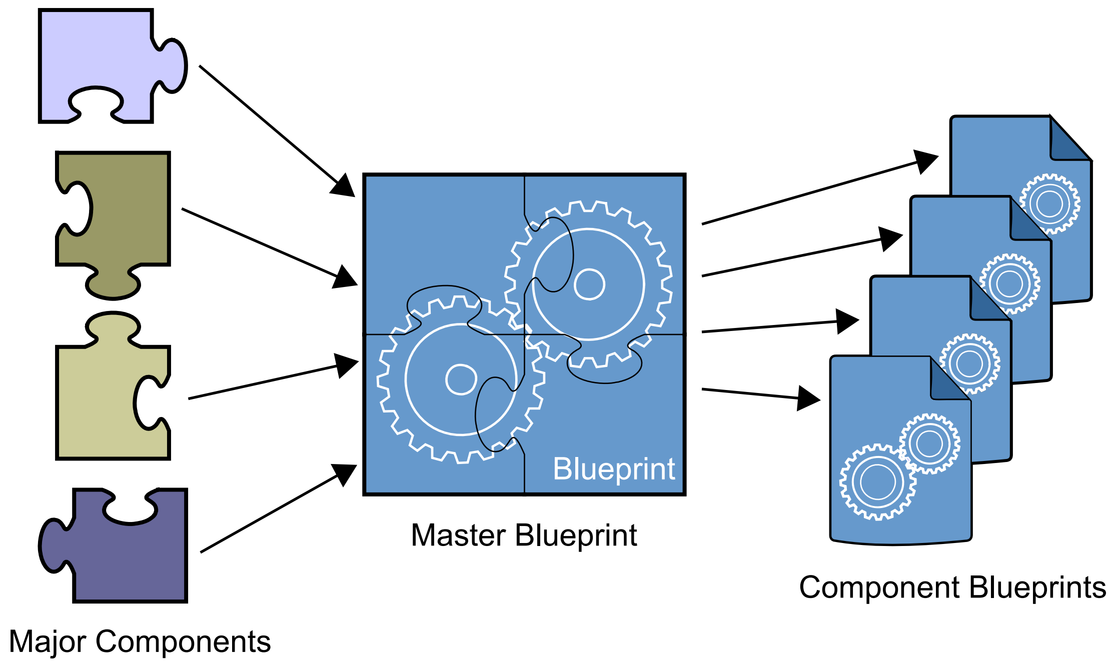
Product Life Cycle (PLC) Stages

1. Analysis
2. Design
3. Development
4. Testing
5. Implementation
6. Maintenance
7. End-of-life (EOL)

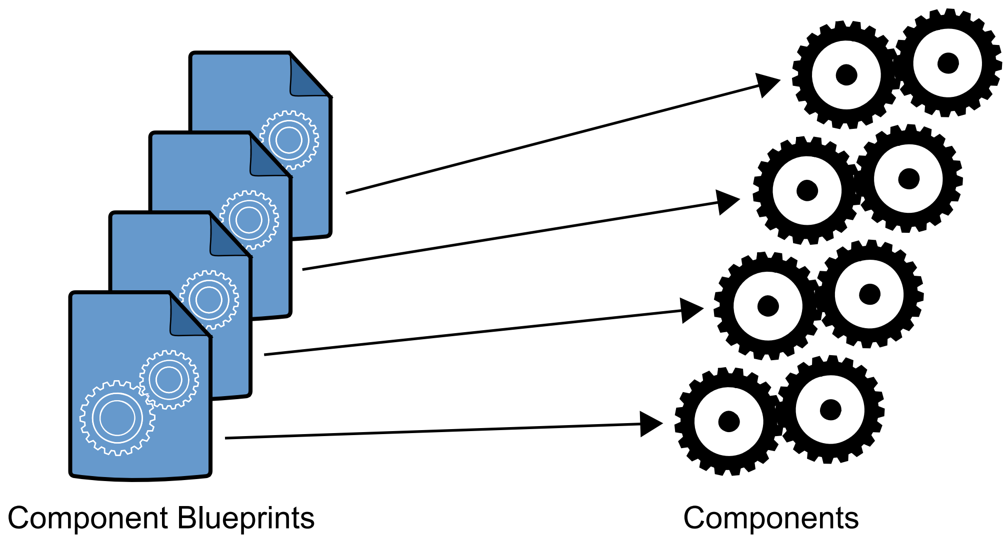
Analysis Stage



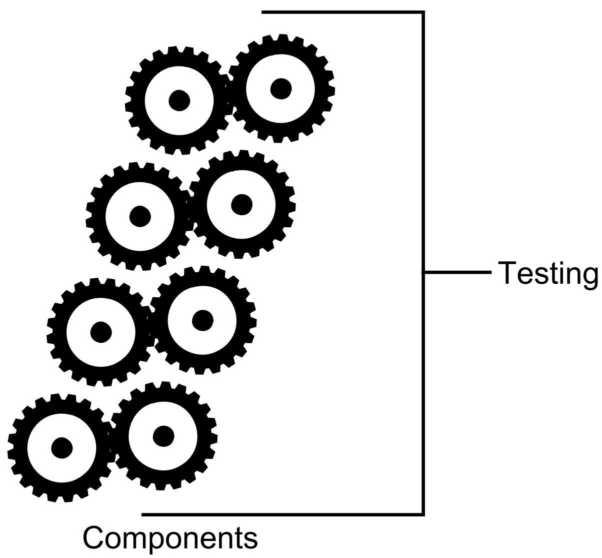
Design Stage



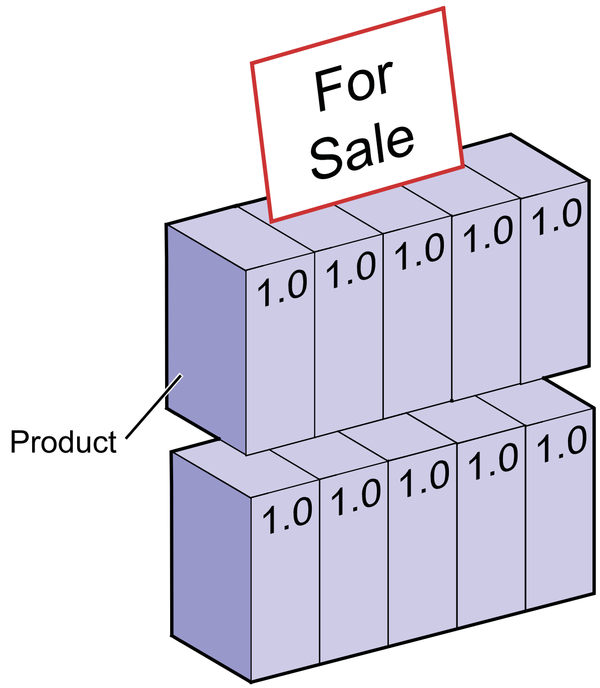
Development Stage



Testing Stage

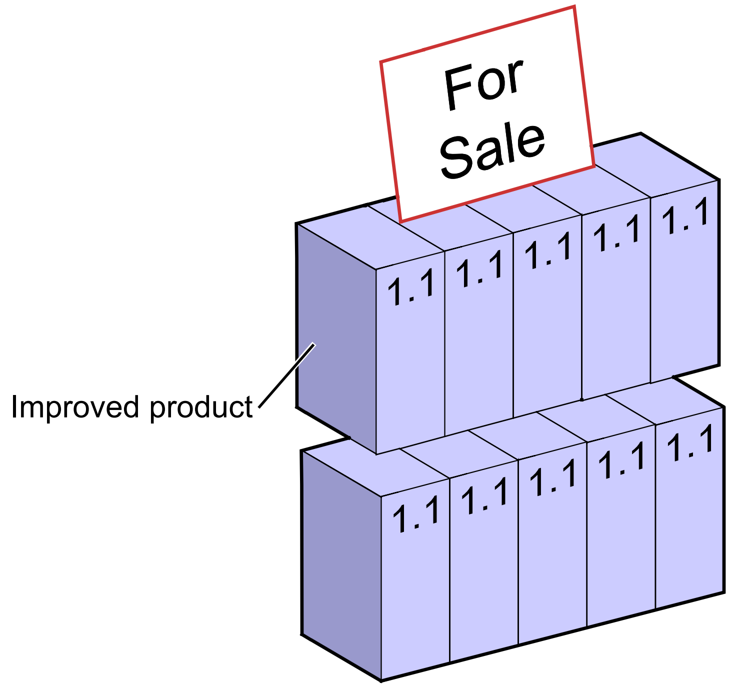


Implementation Stage



Implementation refers to shipping a product such that customers can purchase it.

Maintenance Stage



End-of-Life (EOL) Stage

Module 2

Analyzing a Problem and Designing a Solution

Objectives

- Analyze a problem using object-oriented analysis (OOA)
- Design classes from which objects will be created

Relevance

- How do you decide what components are needed for something you are going to build, such as a house or a piece of furniture?
- What is a taxonomy?
- How are organisms in a taxonomy related?
- What is the difference between attributes and values?

Analyzing a Problem Using OOA

DirectClothing, Inc. sells shirts from their catalog. Business is growing 30 percent per year, and they need a new order entry system.

- DirectClothing produces a catalog of clothing every six months and mails it to subscribers. Each shirt in the catalog has an item identifier (ID), one or more colors (each with a color code), one or more sizes, a description, and a price.
- DirectClothing accepts checks and all credit cards.
- Customers can call DirectClothing to order directly from a customer service representative (CSR), or customers can mail or fax an order form to DirectClothing.

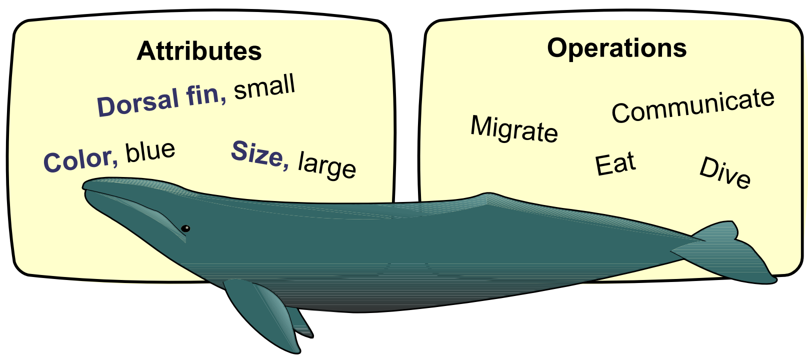
Identifying a Problem Domain

- A problem domain is the scope of the problem you will solve.
- For example, “Create a system allowing order entry people to enter and accept payment for an order.”

Identifying Objects

- Objects can be physical or conceptual.
- Objects have *attributes* (characteristics), such as size, name, shape, and so on.
- Objects have *operations (*the things they can do), such as setting a value, displaying a screen, or increasing speed.

Identifying Objects



Additional Criteria for Recognizing Objects

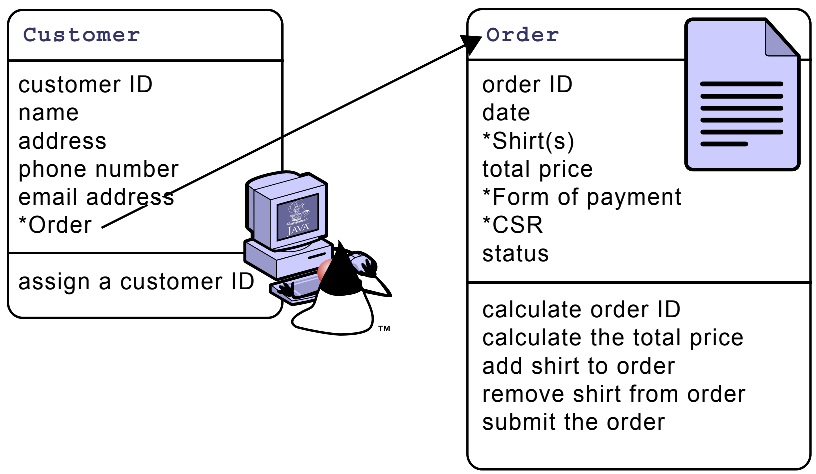
- Relevance to the problem domain:
  - Does the object exist within the boundaries of the problem domain?
  - Is the object required for the solution to be complete?
  - Is the object required as part of an interaction between a user and the system?
- Independent existence

Possible Objects in the DirectClothing, Inc. Case Study

Identifying Object Attributes and Operations

- Attributes are data, such as:
  - ID
  - Order object
- Operations are actions, such as:
  - Delete item
  - Change ID

Object With Another Object as an Attribute



Possible Attributes and Operations for Objects in the DirectClothing, Inc. Case Study

Order

order ID
date
*Shirt(s)
total price
*Form of payment
*CSR
status

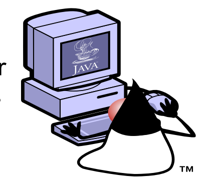
calculate order ID
calculate the total price
add shirt to order
remove shirt from order
submit the order

Shirt

shirtID
price
description
size
color code

calculate shirt ID
display shirt Information

Customer



customer ID
name
address
phone number
email address
*Order

assign a customer ID

Case Study Solution

Order	Shirt
order ID date *Shirt(s) total price *Form of payment *CSR status	shirt ID price description size color code

Order	Shirt
calculate order ID calculate the total price add shirt to order remove shirt from order submit the order	calculate shirt ID display shirt information

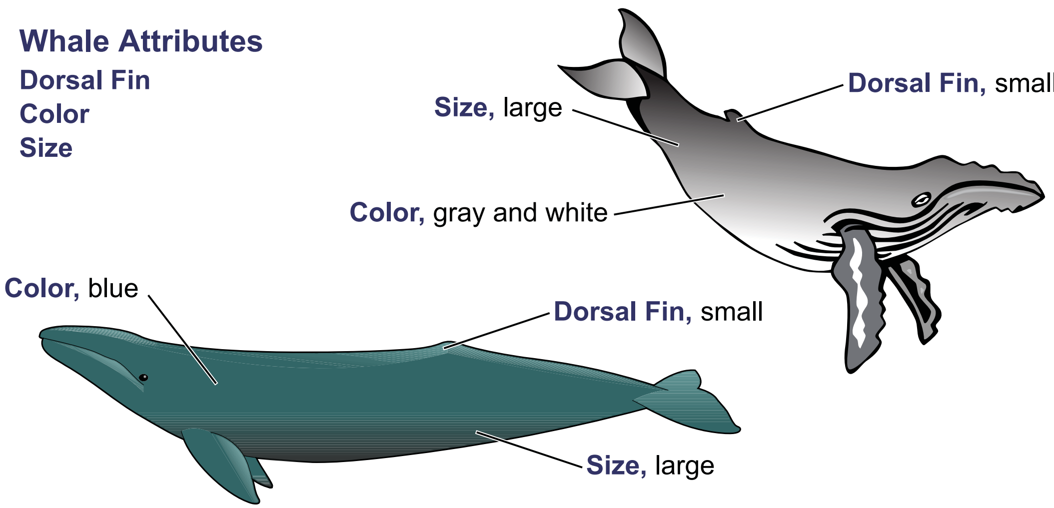
Case Study Solution

Customer	Form of Payment
customer ID name address phone number email address *Order	check number credit card number expiration date
assign a customer ID	verify credit card number verify check payment

Case Study Solution

Catalog	CSR
*Shirt(s)	name extension
add a shirt remove a shirt	

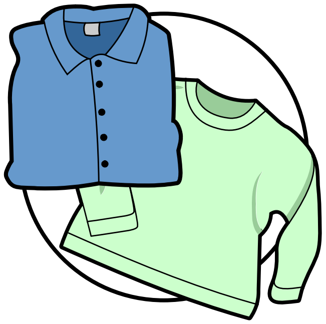
Designing Classes



Class and Resulting Objects

Shirt
shirtID price description size colorCode R=Red, B=Blue, G=Green
calculateShirtID() displayShirtInformation()

`Shirt` Class



`Shirt` Objects

Modeling Classes

- Syntax

ClassName
attributevariableName [*range of values*] attributevariableName [*range of values*] attributevariableName [*range of values*] ...
methodName () *methodName ()* *methodName ()* ...

- Example

Shirt
shirtID price description size colorCode R=Red, B=Blue, G=Green
calculateShirtID() displayInformation()

Module 3

Developing and Testing a Java Technology Program

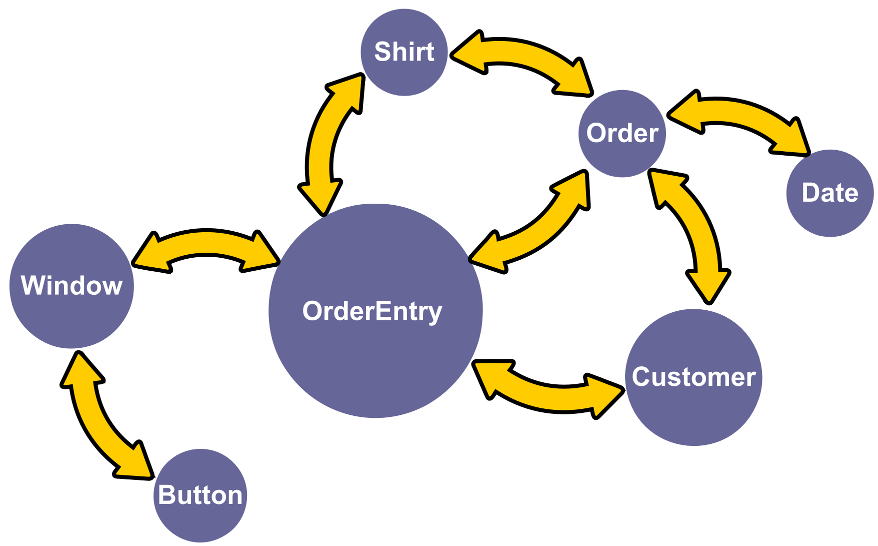
Objectives

- Identify the four components of a class in the Java programming language
- Use the `main` method in a test class to run a Java technology program from the command line
- Compile and execute a Java technology program

Relevance

- How do you test something you have built, such as a house, a piece of furniture, or a program?

Identifying the Components of a Class



Structuring Classes

- The class declaration
- Attribute variable declarations and initialization (optional)
- Methods (optional)
- Comments (optional)

Structuring Classes

```
public class Shirt {

public int shirtID = 0; // Default ID for the shirt
  public String description = "-description required-"; // default
// The color codes are R=Red, B=Blue, G=Green, U=Unset
  public char colorCode = 'U';
  public double price = 0.0; // Default price for all shirts
  public int quantityInStock = 0; // Default quantity for all shirts

// This method displays the values for an item
  public void displayInformation() {
    System.out.println("Shirt ID: " + shirtID);
    System.out.println("Shirt description:" + description);
    System.out.println("Color Code: " + colorCode);
    System.out.println("Shirt price: " + price);
    System.out.println("Quantity in stock: " + quantityInStock);

} // end of display method
} // end of class
```

Class Declaration

- Syntax:

```
[modifiers] class class_identifier
```

- Example:

```
public class Shirt
```

Variable Declarations and Assignments

```
public int shirtID = 0;
public String description = “-description required-”;
public char colorCode = ‘U’;
public double price = 0.0;
public int quantityInStock = 0;
```

Comments

- Single-line:

```
public int shirtID = 0; // Default ID for the shirt
public double price = 0.0; // Default price for all shirts

// The color codes are R=Red, B=Blue, G=Green
```

- Traditional:

```
/********************************************
* Attribute Variable Declaration Section     *
********************************************/
```

Methods

- Syntax:

```
[modifiers] return_type method_identifier ([arguments]){
    method_code_block
}
```

- Example:

```
public void displayInformation() {

    System.out.println("Shirt ID: " + shirtID);
    System.out.println("Shirt description:" + description);
    System.out.println("Color Code: " + colorCode);
    System.out.println("Shirt price: " + price);
    System.out.println("Quantity in stock: " + quantityInStock);

  } // end of display method
```

Creating and Using a Test Class

Example:

```
class ShirtTest {

  public static void main (String args[]) {

    Shirt myShirt;
    myShirt= new Shirt();

    myShirt.displayInformation();


  }
}

```

The `main` Method

- A special method that the JVM recognizes as the starting point for every Java technology program that runs from a command line
- Syntax:

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

Compiling a Program

1. Go the directory where the source code files are stored.
2. Enter the following command for each `.java` file you want to compile.
   - Syntax:

```
javac filename
```

   - Example:

```
javac Shirt.java
```

Executing (Testing) a Program

1. Go the directory where the class files are stored.
2. Enter the following for the class file that contains the `main` method:
   - Syntax

```
java classname
```

   - Example

```
java ShirtTest
```

   - Output:

```
Shirt ID: 0
Shirt description:-description required-
Color Code: U
Shirt price: 0.0
Quantity in stock: 0
```

Debugging Tips

- Error messages state the line number where the error occurs. That line might not always be the actual source of the error.
- Be sure that you have a semicolon at the end of every line where one is required, and no others.
- Be sure that you have an even number of braces.
- Be sure that you have used consistent indentation in your program, as shown in examples in this course.

Module 4

Declaring, Initializing, and Using Variables

Objectives

- Identify the uses for variables and define the syntax for a variable
- List the eight Java programming language primitive data types
- Declare, initialize, and use variables and constants according to Java programming language guidelines and coding standards
- Modify variable values using operators
- Use promotion and type casting

Relevance

- A variable refers to something that can change. Variables can contain one of a set of values. Where have you seen variables before?
- What types of data do you think variables can hold?

Identifying Variable Use and Syntax

Example:

```
public class Shirt {

  public int shirtID = 0; // Default ID for the shirt

  public String description = "-description required-"; // default

  // The color codes are R=Red, B=Blue, G=Green, U=Unset
  public char colorCode = 'U';

  public double price = 0.0; // Default price for all shirts

  public int quantityInStock = 0; // Default quantity for all shirts

// This method displays the values for an item
  public void displayInformation() {

    System.out.println("Shirt ID: " + shirtID);
```

Identifying Variable Use and Syntax

Example (continued)

```
        System.out.println("Shirt description:" + description);
        System.out.println("Color Code: " + colorCode);
        System.out.println("Shirt price: " + price);
        System.out.println("Quantity in stock: " + quantityInStock);

    } // end of display method

} // end of class
```

Uses for Variables

- Holding unique data for an object instance
- Assigning the value of one variable to another
- Representing values within a mathematical expression
- Printing the values to the screen
- Holding references to other objects

Variable Declaration and Initialization

- Syntax (attribute or instance variables):

```
[modifiers] type identifier [= value];
```

- Syntax (local variables):

```
type identifier;
```

- Syntax (local variables)

```
type identifier [= value];
```

- Examples:

```
public int shirtID = 0;
public String description = "-description required-";
public char colorCode = 'U';
public double price = 0.0;
public int quantityInStock = 0;
```

Describing Primitive Data Types

- Integral types (`byte`, `short`, `int`, and `long`)
- Floating point types (`float` and `double`)
- Textual type (`char`)
- Logical type (`boolean`)

Integral Primitive Types

Type	Length	Range	Examples of Allowed Literal Values
`byte`	8 bits	-2^{7} to 2^{7} -1 (-128 to 127, or 256 possible values)	2 -114
`short`	16 bits	-2^{15} to 2^{15} -1 (-32,768 to 32,767, or 65,535 possible values)	2 -32699
`int`	32 bits	-2^{31} to 2^{31} -1 (-2,147,483,648 to 2,147,483,647 or 4,294,967,296 possible values)	2 147334778

Integral Primitive Types

Type	Length	Range	Examples of Allowed Literal Values
`long`	64 bits	-2^{63} to 2^{63} -1 (-9,223,372,036854,775,808 to 9,223,372,036854,775,807, or 18,446,744,073,709,551,616 possible values)	2 -2036854775808L 1L

Integral Primitive Types

```
public int shirtID = 0; // Default ID for the shirt
public int quantityInStock = 0; // Default quantity for all shirts
```

Floating Point Primitive Types

Type	Float Length	Examples of Allowed Literal Values
`float`	32 bits	99F -327456,99.01F 4.2E6F (engineering notation for $4.2 * 10^6$)
`double`	64 bits	-1111 2.1E12 99970132745699.999

```
public double price = 0.0; // Default price for all shirts
```

Textual Primitive Type

- The only data type is `char`
- Used for a single character (16 bits)
- Example:

```
public char colorCode = 'U';
```

Logical Primitive Type

- The only data type is `boolean`
- Can store only `true` or `false`
- Holds the result of an expression that evaluates to either `true` or `false`

Naming a Variable

Rules:

- Variable identifiers must start with either an uppercase or lowercase letter, an underscore (_), or a dollar sign ($).
- Variable identifiers cannot contain punctuation, spaces, or dashes.
- Java technology keywords cannot be used.

Naming a Variable

Guidelines:

- Begin each variable with a lowercase letter; subsequent words should be capitalized, such as `myVariable`.
- Choose names that are mnemonic and that indicate to the casual observer the intent of the variable.

Assigning a Value to a Variable

- Example:

```
double price = 12.99;
```

- Example (`boolean`):

```
boolean isOpen = false;
```

Declaring and Initializing Several Variables in One Line of Code

- Syntax:

```
type identifier = value [, identifier = value];
```

- Example:

```
double price = 0.0, wholesalePrice = 0.0;
```

Additional Ways to Declare Variables and Assign Values to Variables

- Assigning literal values:

```
int ID = 0;
float pi = 3.14F;
char myChar = 'G';
boolean isOpen = false;
```

- Assigning the value of one variable to another variable:

```
int ID = 0;
int saleID = ID;
```

Additional Ways to Declare Variables and Assign Values to Variables

- Assigning the result of an expression to integral, floating point, or Boolean variables

```
float numberOrdered = 908.5F;
float casePrice = 19.99F;
float price = (casePrice * numberOrdered);

int hour = 12;
boolean isOpen = (hour > 8);
```

- Assigning the return value of a method call to a variable

Constants

- Variable (can change):

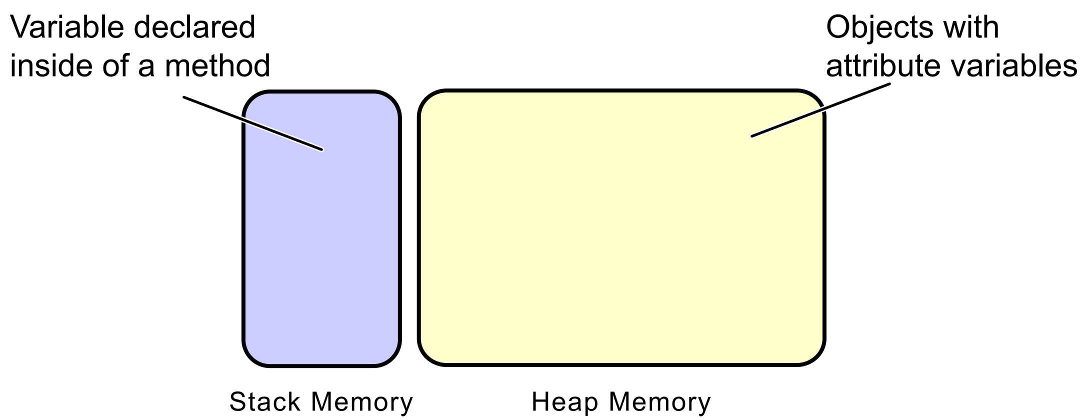
```
double salesTax = 6.25;
```

- Constant (cannot change):

```
final double SALES_TAX = 6.25;
```

- Guideline – Constants should be capitalized with words separated by an underscore (_).

Storing Primitives and Constants in Memory



Standard Mathematical Operators

Purpose	Operator	Example	Comments
Addition	+	`sum = num1 + num2;` If `num1` is 10 and `num2` is 2, `sum` is 12.	
Subtraction	–	`diff = num1 – num2;` If `num1` is 10 and `num2` is 2, `diff` is 8.	
Multiplication	*	`prod = num1 * num2;` If `num1` is 10 and `num2` is 2, `prod` is 20.	
Division	/	`quot = num1 / num2;` If `num1` is 31 and `num2` is 6, `quot` is 5.	Division returns an integer value (with no remainder).

Standard Mathematical Operators

Purpose	Operator	Example	Comments
Remainder	`%`	`mod = num1 % num2;` If `num1` is 31 and `num2` is 6, `mod` is 1.	Remainder finds the remainder of the first number divided by the second number. 5 R1 6 ⌈ 31 30 1 Remainder always gives an answer with the same sign as the first operand.

Increment and Decrement Operators (++ and --)

The long way:

```
age = age + 1;
```

Increment and Decrement Operators (++ and --)

The short way:

Operator	Purpose	Example	Notes

++	Pre-increment (++*variable*)	`int i = 6;`
		`int j = ++i;`
		`i is 7, j is 7`	

	Post-increment (*variable*++)	`int i = 6;`
		`int j = i++;`
		`i is 7, j is 6`
			The value of `i` is assigned to `j` before `i` is incremented. Therefore, `j` is assigned `6`.

Increment and Decrement Operators (++ and --)

Operator	Purpose	Example	Notes

`--`	Pre-decrement (`--`*`variable`*)	`int i = 6;`
		`int j = --i;`
		`i is 5, j is 5`	

	Post-decrement (*`variable`*`--`)	`int i = 6;`
		`int j = i--;`
		`i is 5, j is 6`
			The value `i` is assigned to `j` before `i` is decremented. Therefore, `j` is assigned 6.

Increment and Decrement Operators (++ and – –)

Examples:

```
int count=15;
int a, b, c, d;
a = count++;
b = count;
c = ++count;
d = count;
System.out.println(a + ", " + b + ", " + c + ", " + d);
```

Operator Precedence

Rules of precedence:

1. Operators within a pair of parentheses
2. Increment and decrement operators
3. Multiplication and division operators, evaluated from left to right
4. Addition and subtraction operators, evaluated from left to right

Example of need for rules of precedence (is the answer 34 or 9?):

```
c = 25 - 5 * 4 / 2 - 10 + 4;
```

Using Parentheses

Examples:

```
c = (((25 - 5) * 4) / (2 - 10)) + 4;
c = ((20 * 4) / (2 - 10)) + 4;
c = (80 / (2 - 10)) + 4;
c = (80 / -8) + 4;
c = -10 + 4;
c = -6;
```

Using Promotion and Type Casting

- Example of potential issue:

```
int num1 = 53; // 32 bits of memory to hold the value
int num2 = 47; // 32 bits of memory to hold the value
byte num3; // 8 bits of memory reserved
num3 = (num1 + num2); // causes compiler error
```

- Example of potential solution:

```
int num1 = 53;
int num2 = 47;
long num3;
num3 = (num1 + num2);
```

Promotion

- Automatic promotions:
  - If you assign a smaller type to a larger type
  - If you assign an integral type to a floating point type
- Examples of automatic promotions:

```
long big = 6;
```

Type Casting

- Syntax:

```
identifier = (target_type) value
```

- Example of potential issue:

```
int num1 = 53; // 32 bits of memory to hold the value
int num2 = 47; // 32 bits of memory to hold the value
byte num3; // 8 bits of memory reserved
num3 = (num1 + num2); // causes compiler error
```

- Example of potential solution:

```
int num1 = 53; // 32 bits of memory to hold the value
int num2 = 47; // 32 bits of memory to hold the value
byte num3; // 8 bits of memory reserved
num3 = (byte)(num1 + num2); // no data loss
```

Type Casting

Examples:

```
int myInt;
long myLong = 99L;
myInt = (int) (myLong); // No data loss, only zeroes.
                        // A much larger number would
                        // result in data loss.

int myInt;
long myLong = 123987654321L;
myInt = (int) (myLong); // Number is "chopped"
```

Compiler Assumptions for Integral and Floating Point Data Types

- Example of potential problem:

```
short a, b, c;
a = 1 ;
b = 2 ;
c = a + b ; //compiler error
```

- Example of potential solutions:
  - Declare `c` as an `int` type in the original declaration:

```
int c;
```

  - Type cast the (`a+b`) result in the assignment line:

```
c = (short)(a+b);
```

Floating Point Data Types and Assignment

- Example of potential problem:

```
float float1 = 27.9;//compiler error
```

- Example of potential solutions:
  - The `F` notifies the compiler that 27.9 is a `float` value:

```
float float1 = 27.9F;
```

  - `27.9` is cast to a `float` type:

```
float float1 = (float) 27.9;
```

Example

```
public class Person {

  public int ageYears = 32;

  public void calculateAge() {

    int ageDays = ageYears * 365;
    long ageSeconds = ageYears * 365 * 24L * 60 * 60;

    System.out.println("You are " + ageDays + " days old.");
    System.out.println("You are " + ageSeconds + " seconds old.");

  } // end of calculateAge method
} // end of class
```

Module 5

Creating and Using Objects

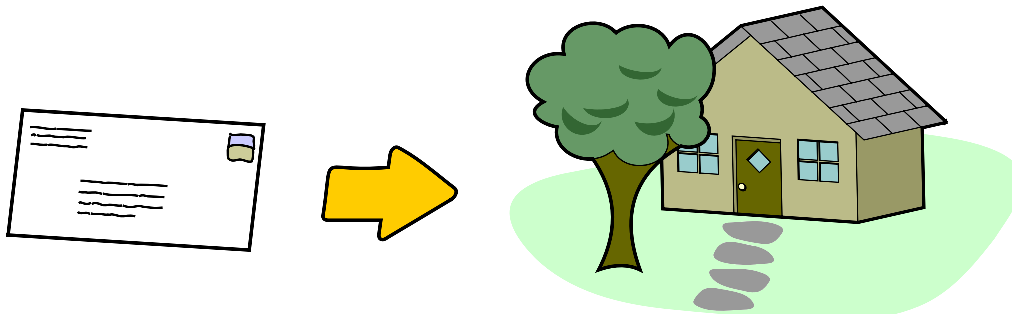
Objectives

- Declare, instantiate, and initialize object reference variables
- Compare how object reference variables are stored in relation to primitive variables
- Use a class (the `String` class) included in the Java SDK
- Use the Java Platform, Standard Edition (Java SE™) class library specification to learn about other classes in this application programming interface (API)

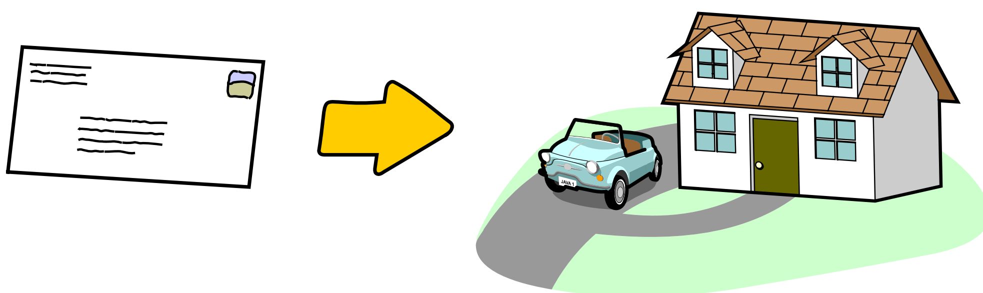
Relevance

- What does it mean to create an instance of a blueprint for a house?
- How do you refer to different houses on the same street?
- When a builder builds a house, does the builder build every component of the house, including the windows, doors, and cabinets?

Declaring Object References, Instantiating Objects, and Initializing Object References



3222 Jones St.



777 Boulder Ln.

Declaring Object References, Instantiating Objects, and Initializing Object References

Example:

```
class ShirtTest {

public static void main (String args[]) {

    Shirt myShirt = new Shirt();

    myShirt.displayInformation();

  }
}
```

Declaring Object Reference Variables

- Syntax:

  Classname identifier;

- Example:

  ```
  Shirt myShirt;
  ```

Instantiating an Object

Syntax:

```
new Classname()
```

Initializing Object Reference Variables

- The assignment operator
- Example:

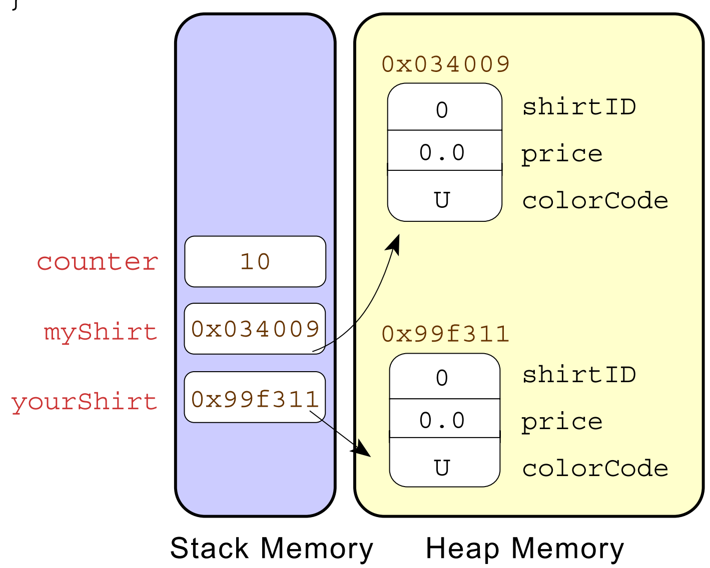
```
myShirt = new Shirt();
```

Using an Object Reference Variable to Manipulate Data

```
public class ShirtTestTwo {

  public static void main (String args[]) {

    Shirt myShirt = new Shirt();
    Shirt yourShirt = new Shirt();

    myShirt.displayInformation();
    yourShirt.displayInformation();

    myShirt.colorCode='R';
    yourShirt.colorCode='G';

    myShirt.displayInformation();
    yourShirt.displayInformation();

  }
}
```

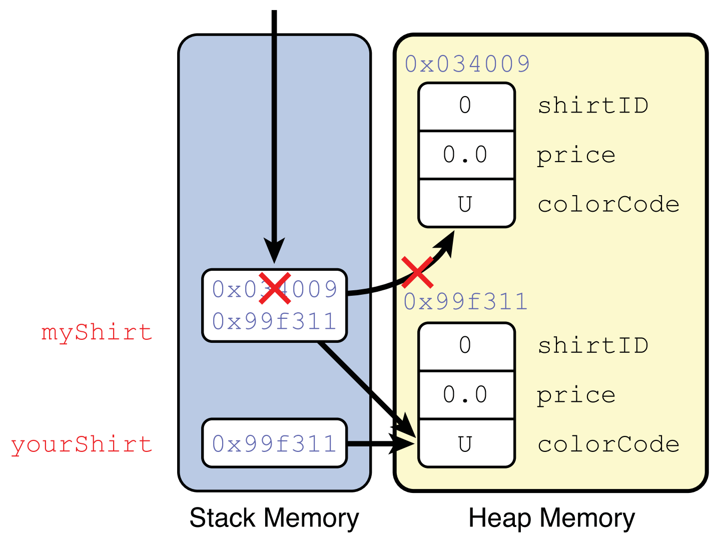
Storing Object Reference Variables in Memory

```
public static void main (String args[]) {

  int counter;
  counter = 10;
  Shirt myShirt = new Shirt ( );
}
```



Assigning an Object Reference From One Variable to Another

```
Shirt  myShirt - new Shirt( );
Shirt  yourShirt = new Shirt( );
myShirt = yourShirt;
```



Using the `String` Class

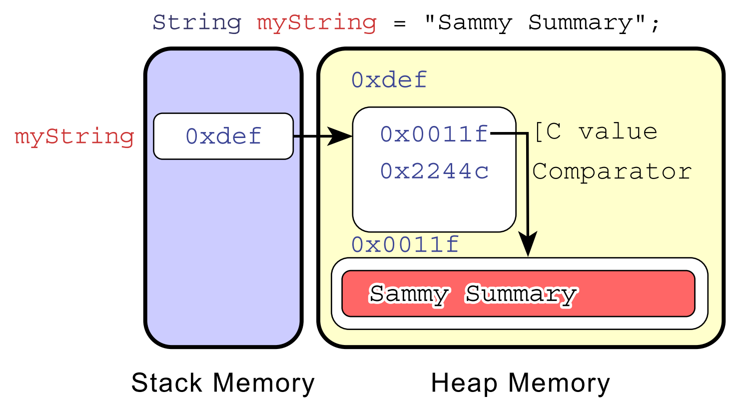
- Creating a `String` object with the `new` keyword:

```
String myName = new String("Fred Smith");
```

- Creating a `String` object without the `new` keyword:

```
String myName = "Fred Smith";
```

Storing `String` Objects in Memory



Using Reference Variables for `String` Objects

Example:

```
public class PersonTwo {

  public String name = "Jonathan";
  public String job = "Ice Cream Taster";

  public void display(){
    System.out.println("My name is " + name + ", I am a " + job);
  }
} // end of class
```

Investigating the Java Class Libraries

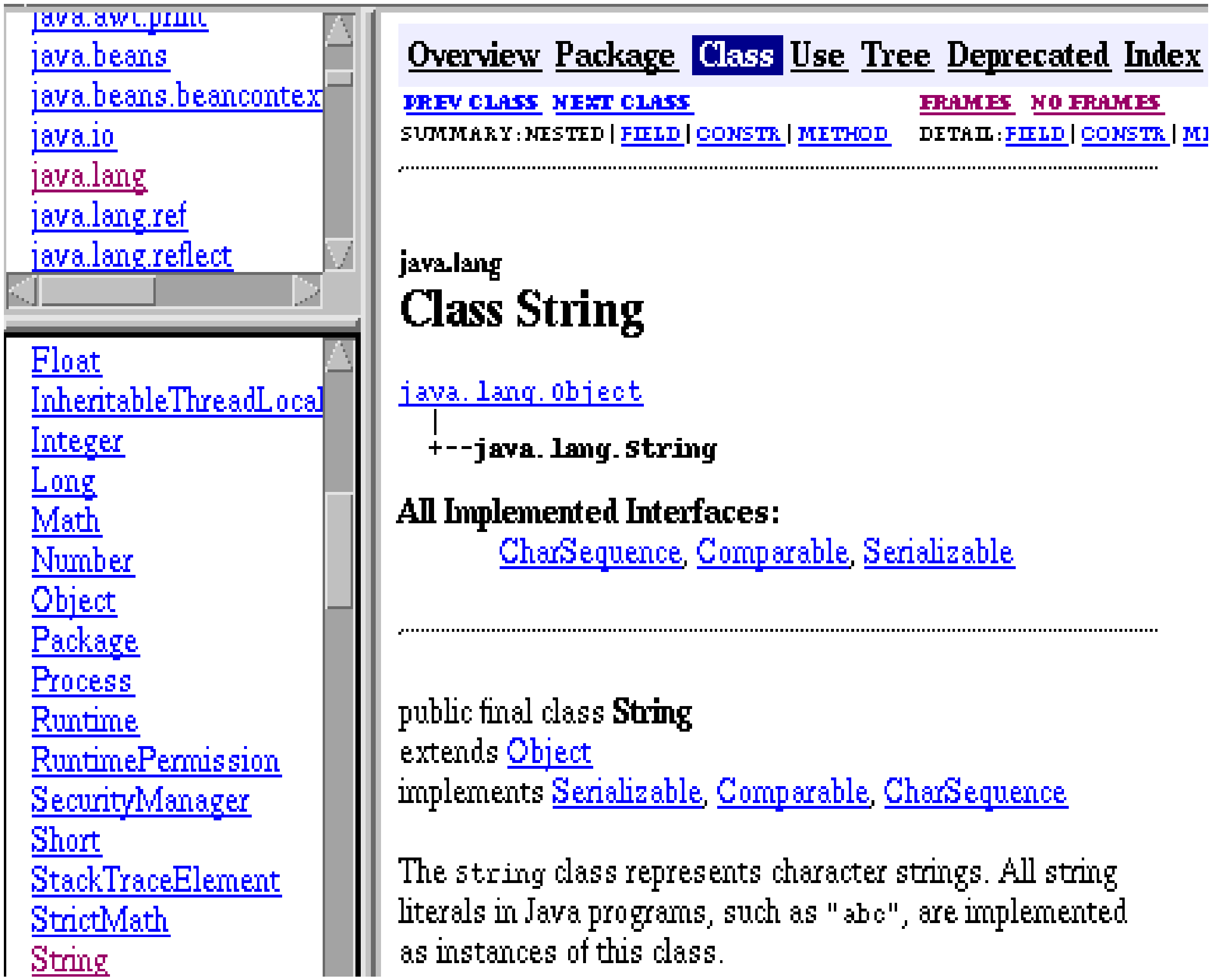
- Universal Resource Locator (URL) to view the Java SE specification:

`http://java.sun.com/reference/api/`

- Example:

`http://java.sun.com/javase/6/docs/api/`

Investigating the Java Class Libraries



Using the Java Class Library Specification to Learn About a Method

- The `println` method:

```
System.out.println(data_to_print_to_the_screen);
```

- Example:

```
System.out.print("Carpe diem");
System.out.println("Seize the day");
```

prints this:

```
Carpe diem Seize the day
```

Module 6

Using Operators and Decision Constructs

Objectives

- Identify relational and conditional operators
- Create `if` and `if/else` constructs
- Use the `switch` constructs

Relevance

- When you must make a decision that has several different paths, how do you ultimately choose one path over all the other paths?
- For example, what are all of the things that go through your mind when you are going to purchase an item?

Using Relational and Conditional Operators

Elevator Example

```
public class Elevator {

  public boolean doorOpen=false; // Doors are closed by default
  public int currentFloor = 1; // All elevators start on first floor
  public final int TOP_FLOOR = 10;
  public final int MIN_FLOORS = 1;

  public void openDoor() {
    System.out.println("Opening door.");
    doorOpen = true;
    System.out.println("Door is open.");
  }

  public void closeDoor() {
    System.out.println("Closing door.");
    doorOpen = false;
    System.out.println("Door is closed.");
  }
```

Elevator Example

```

  public void goUp() {
    System.out.println("Going up one floor.");
    currentFloor++;
    System.out.println("Floor: " + currentFloor);
  }

  public void goDown() {
    System.out.println("Going down one floor.");
    currentFloor--;
    System.out.println("Floor: " + currentFloor);
  }


}
```

The `ElevatorTest.java` File

```
public class ElevatorTest {
  public static void main(String args[]) {

    Elevator myElevator = new Elevator();

     myElevator.openDoor();
     myElevator.closeDoor();
     myElevator.goDown();
     myElevator.goUp();
     myElevator.goUp();
     myElevator.goUp();
     myElevator.openDoor();
     myElevator.closeDoor();
     myElevator.goDown();
     myElevator.openDoor();
     myElevator.goDown();
     myElevator.openDoor();
   }
}
```

Relational Operators

Condition	Operator	Example
Is equal to	`==`	`int i=1;` `(i == 1)`
Is not equal to	`!=`	`int i=2;` `(i != 1)`
Is less than	`<`	`int i=0;` `(i < 1)`
Is less than or equal to	`<=`	`int i=1;` `(i <= 1)`
Is greater than	`>`	`int i=2;` `(i > 1)`
Is greater than or equal to	`>=`	`int i=1;` `(i >= 1)`

Testing Equality Between Strings

Example:

```
public class Employees {

  public String name1 = "Fred Smith";
  public String name2 = "Joseph Smith";

  public void areNamesEqual() {

    if (name1.equals(name2)) {
      System.out.println("Same name.");
    }
    else {
      System.out.println("Different name.");
    }
  }
}

```

Common Conditional Operators

Operation	Operator	Example

If one condition AND another condition	`&&`	`int i = 2;`
		`int j = 8;`
		`((i < 1) && (j > 6))`

If either one condition OR another condition	`||`	`int i = 2;`
		`int j = 8;`
		`((i < 1) || (j > 10))`

NOT	`!`	`int i = 2;` `(!(i < 3))`

The `if` Construct

- Syntax:

```
if (boolean_expression) {
   code_block;
}   // end of if construct
// program continues here
```

- Example of potential output:

```
Opening door.
Door is open.
Closing door.
Door is closed.
Going down one floor.
Floor: 0 <--- this is a error in logic
Going up one floor.
Floor: 1
Going up one floor.
Floor: 2
...
```

The `if` Construct

Example of potential solution:

```

public class IfElevator {

  public boolean doorOpen=false; // Doors are closed by default
  public int currentFloor = 1; // All elevators start on first floor
  public final int TOP_FLOOR = 10;
  public final int MIN_FLOORS = 1;

  public void openDoor() {
    System.out.println("Opening door.");
    doorOpen = true;
    System.out.println("Door is open.");
  }
  public void closeDoor() {
    System.out.println("Closing door.");
    doorOpen = false;
```

The `if` Construct

```
System.out.println(“Door is closed.”);
  }
  public void goUp() {
    System.out.println(“Going up one floor.”);
    currentFloor++;
    System.out.println(“Floor: “ + currentFloor);
  }
  public void goDown() {

    if (currentFloor == MIN_FLOORS) {
      System.out.println(“Cannot Go down”);
    }
    if (currentFloor > MIN_FLOORS) {
      System.out.println(“Going down one floor.”);
      currentFloor--;
      System.out.println(“Floor: “ + currentFloor);
    }
  }
}
```

The `if` Construct

Example potential output:

```
Opening door.
Door is open.
Closing door.
Door is closed.
Cannot Go down <--- elevator logic prevents problem
Going up one floor.
Floor: 2
Going up one floor.
Floor: 3
...
```

Nested `if` Statements

Example:

```
public class NestedIfElevator {

  public boolean doorOpen=false; // Doors are closed by default
  public int currentFloor = 1; // All elevators start on first floor
  public final int TOP_FLOOR = 10;
  public final int MIN_FLOORS = 1;

  public void openDoor() {
    System.out.println(“Opening door.”);
    doorOpen = true;
    System.out.println(“Door is open.”);
  }

  public void closeDoor() {
    System.out.println(“Closing door.”);
    doorOpen = false;
```

Nested `if` Statements

```
System.out.println(“Door is closed.”);
  }

  public void goUp() {
    System.out.println(“Going up one floor.”);
    currentFloor++;
    System.out.println(“Floor: “ + currentFloor);
  }

  public void goDown() {

    if (currentFloor == MIN_FLOORS) {
      System.out.println(“Cannot Go down”);
    }

    if (currentFloor > MIN_FLOORS) {

      if (!doorOpen) {

```

Nested `if` Statements

```
            System.out.println(“Going down one floor.”);
            currentFloor--;
            System.out.println(“Floor: “ + currentFloor);
          }
        }
      }


    }

```

The `if/else` Construct

Syntax:

```
if (boolean_expression) {

     code_block;

} // end of if construct

else {

     code_block;

} // end of else construct

// program continues here
```

The `if/else` Construct

Example:

```
public class IfElseElevator {

  public boolean doorOpen=false; // Doors are closed by default
  public int currentFloor = 1; // All elevators start on first floor
  public final int TOP_FLOOR = 10;
  public final int MIN_FLOORS = 1;

  public void openDoor() {
    System.out.println("Opening door.");
    doorOpen = true;
    System.out.println("Door is open.");
  }
  public void closeDoor() {
    System.out.println("Closing door.");
    doorOpen = false;
    System.out.println("Door is closed.");
  }
```

The `if/else` Construct

```

  public void goUp() {
    System.out.println(“Going up one floor.”);
    currentFloor++;
    System.out.println(“Floor: “ + currentFloor);
  }

  public void goDown() {

    if (currentFloor == MIN_FLOORS) {
      System.out.println(“Cannot Go down”);
    }
    else {
      System.out.println(“Going down one floor.”);
      currentFloor--;
      System.out.println(“Floor: “ + currentFloor);}
    }
  }
}
```

The `if/else` Construct

Example potential output:

```
Opening door.
Door is open.
Closing door.
Door is closed.
Cannot Go down <--- elevator logic prevents problem
Going up one floor.
Floor: 2
Going up one floor.
Floor: 3
...
```

Chaining `if/else` Constructs

Syntax:

```
if (boolean_expression) {

     code_block;

} // end of if construct

else if (boolean_expression){

     code_block;

} // end of else if construct

else {

     code_block;
}
// program continues here
```

Chaining `if/else` Constructs

Example:

```
public class IfElseDate {

  public int month = 10;

  public void calculateNumDays() {

    if (month == 1 || month == 3 || month == 5 || month == 7 ||
  month == 8 || month == 10 || month == 12) {

      System.out.println(“There are 31 days in that month.”);
    }

    else if (month == 2) {
      System.out.println(“There are 28 days in that month.”);
    }

```

Chaining `if/else` Constructs

```
else if (month == 4 || month == 6 || month == 9 || month == 11) {
      System.out.println(“There are 30 days in that month.”);
    }

    else {
      System.out.println(“Invalid month.”);
    }
  }
}

```

Using the `switch` Construct

Syntax:

```
switch (variable) {
    case literal_value:
        code_block;
        [break;]
    case another_literal_value:
        code_block;
        [break;]
    [default:]
        code_block;
}
```

Using the `switch` Construct

Example:

```

public class SwitchDate {

  public int month = 10;

  public void calculateNumDays() {

    switch(month) {
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12:
      System.out.println("There are 31 days in that month.");
      break;
```

Using the `switch` Construct

```
case 2:
      System.out.println("There are 28 days in that month.");
      break;
    case 4:
    case 6:
    case 9:
    case 11:
      System.out.println("There are 30 days in that month.");
      break;
    default:
      System.out.println("Invalid month.");
      break;
    }
  }
}
```

When to Use `switch` Constructs

- Equality tests
- Tests against a *single* value, such as `customerStatus`
- Tests against the value of an `int`, `short`, `byte`, or `char` type

Module 7

Using Loop Constructs

Objectives

- Create `while` loops
- Develop `for` loops
- Create `do/while` loops

Relevance

What are some situations when you would want to continue performing a certain action, as long as a certain condition existed?

Creating `while` Loops

Syntax:

```
while (boolean_expression) {
   code_block;
}   // end of while construct
// program continues here
```

Creating `while` Loops

Example:

```
public class WhileElevator {

  public boolean doorOpen=false;
  public int currentFloor = 1;

  public final int TOP_FLOOR = 5;
  public final int BOTTOM_FLOOR = 1;

  public void openDoor() {
    System.out.println("Opening door.");
    doorOpen = true;
    System.out.println("Door is open.");
  }

  public void closeDoor() {
    System.out.println("Closing door.");
```

Creating `while` Loops

```
doorOpen = false;
    System.out.println("Door is closed.");
  }

  public void goUp() {
    System.out.println("Going up one floor.");
    currentFloor++;
    System.out.println("Floor: " + currentFloor);
  }

  public void goDown() {
    System.out.println("Going down one floor.");
    currentFloor--;
    System.out.println("Floor: " + currentFloor);
  }

  public void setFloor() {
```

Creating `while` Loops

```
// Normally you would pass the desiredFloor as an argument to the
    // setFloor method. However, because you have not learned how to
    // do this yet, desiredFloor is set to a specific number (5)
    // below.

    int desiredFloor = 5;

    while (currentFloor != desiredFloor){
    if (currentFloor < desiredFloor) {
      goUp();
      }
    else {
      goDown();
      }
    }

}

```

Nested `while` Loops

Example potential solution:

```
public class WhileRectangle {
  public int height = 3;
  public int width = 10;
  public void displayRectangle() {
    int colCount = 0;
    int rowCount = 0;
    while (rowCount < height) {
      colCount=0;
      while (colCount < width) {
        System.out.print("@");
        colCount++;
}
      System.out.println();
      rowCount++;
    }
  }
}
```

Developing a `for` Loop

Syntax:

```
for (initialize[,initialize]; boolean_expression;
     update[,update]) {

     code_block;
}
```

Developing a `for` Loop

Example:

```

public class ForElevator {

  public boolean doorOpen=false;
  public int currentFloor = 1;

  public final int TOP_FLOOR = 5;
  public final int BOTTOM_FLOOR = 1;

  public void openDoor() {
    System.out.println("Opening door.");
    doorOpen = true;
    System.out.println("Door is open.");
  }

  public void closeDoor() {
    System.out.println("Closing door.");
```

Developing a `for` Loop

```
doorOpen = false;
    System.out.println("Door is closed.");
  }

  public void goUp() {
    System.out.println("Going up one floor.");
    currentFloor++;
    System.out.println("Floor: " + currentFloor);
  }

  public void goDown() {
    System.out.println("Going down one floor.");
    currentFloor--;
    System.out.println("Floor: " + currentFloor);
  }

  public void setFloor() {

```

Developing a `for` Loop

```
// Normally you would pass the desiredFloor as an argument to the
    // setFloor method. However, because you have not learned how to
    // do this yet, desiredFloor is set to a specific number (5)
    // below.
    int desiredFloor = 5;

    if (currentFloor > desiredFloor) {
      for (int down = currentFloor; down != desiredFloor; --down) {
        goDown();
      }
    }
    else {
      for (int up = currentFloor; up != desiredFloor; ++up) {
        goUp();
      }
    }
  }
}

```

Nested `for` Loops

Example:

```
public class ForRectangle {

  public int height = 3;
  public int width = 10;

  public void displayRectangle() {

    for (int rowCount = 0; rowCount < height; rowCount++) {
      for (int colCount = 0; colCount < width; colCount++) {
        System.out.print("@");
      }
      System.out.println();
    }
  }
}
```

Coding a `do/while` Loop

Syntax:

```
do {

     code_block;
}
while (boolean_expression);// Semicolon is mandatory.
```

Coding a `do/while` Loop

Example:

```

public class DoWhileElevator {

  public boolean doorOpen=false;
  public int currentFloor = 1;

  public final int TOP_FLOOR = 5;
  public final int BOTTOM_FLOOR = 1;

  public void openDoor() {
    System.out.println("Opening door.");
    doorOpen = true;
    System.out.println("Door is open.");
  }

  public void closeDoor() {
    System.out.println("Closing door.");
```

Coding a do/while Loop

```
doorOpen = false;
      System.out.println("Door is closed.");
    }

    public void goUp() {
      System.out.println("Going up one floor.");
      currentFloor++;
      System.out.println("Floor: " + currentFloor);
    }

    public void goDown() {
      System.out.println("Going down one floor.");
      currentFloor--;
      System.out.println("Floor: " + currentFloor);
    }

    public void setFloor() {

```

Coding a `do/while` Loop

```
// Normally you would pass the desiredFloor as an argument to the
    // setFloor method. However, because you have not learned how to
    // do this yet, desiredFloor is set to a specific number (5)
    // below.

    int desiredFloor = 5;

    do {
      if (currentFloor < desiredFloor) {
        goUp();
      }
    else if (currentFloor > desiredFloor) {
        goDown();
      }
    }
while (currentFloor != desiredFloor);
  }

}
```

Nested `do/while` Loops

Example:

```
public class DoWhileRectangle {

  public int height = 3;
  public int width = 10;

  public void displayRectangle() {

    int rowCount = 0;
    int colCount = 0;

    do {
      colCount = 0;

      do {
        System.out.print("@");
        colCount++;
```

Nested `do/while` Loops

```
}
      while (colCount < width);

      System.out.println();
      rowCount++;
    }
    while (rowCount < height);
  }
}

```

Comparing Loop Constructs

- Use the `while` loop to iterate indefinitely through statements and to perform the statements zero or more times.
- Use the `do/while` loop to iterate indefinitely through statements and to perform the statements *one* or more times.
- Use the `for` loop to step through statements a predefined number of times.

Module 8

Developing and Using Methods

Overview

- Objectives:
  - Describe the advantages of methods and define worker and calling methods
  - Declare and invoke a method
  - Compare object and static methods
  - Use overloaded methods

Relevance

How do you strucure or implement the operations performed on an object?

Creating and Invoking Methods

Syntax:

```
[modifiers] return_type method_identifier ([arguments]) {
   method_code_block
}
```

Basic Form of a Method

Example:

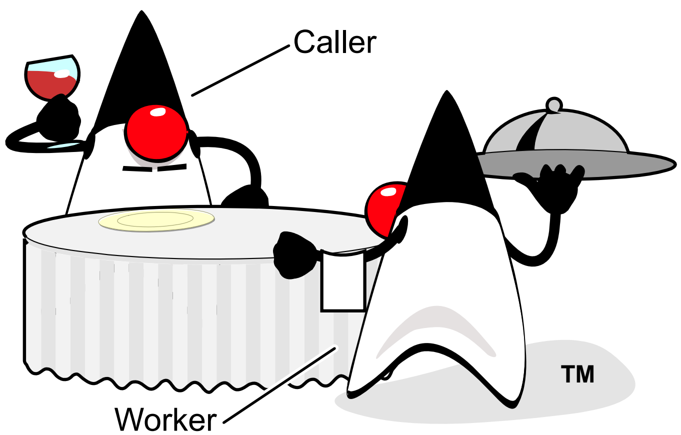
```
public void displayInformation() {
   System.out.println("Shirt ID: " + shirtID);
    System.out.println("Shirt description:" + description);
   System.out.println("Color Code: " + colorCode);
    System.out.println("Shirt price: " + price);
    System.out.println("Quantity in stock: " + quantityInStock);
} // end of display method
```

Invoking a Method From a Different Class

Example:

```

public class ShirtTest {

  public static void main (String args[]) {

  Shirt myShirt;
  myShirt = new Shirt();

  myShirt.displayInformation();


   }
}

```

Calling and Worker Methods



Invoking a Method in the Same Class

Example:

```

public class Elevator {

    public boolean doorOpen=false;
    public int currentFloor = 1;

    public final int TOP_FLOOR = 5;
    public final int BOTTOM_FLOOR = 1;

    public void openDoor() {
      System.out.println(“Opening door.”);
      doorOpen = true;
      System.out.println(“Door is open.”);
    }

    public void closeDoor() {
      System.out.println(“Closing door.”);
```

Invoking a Method in the Same Class

```
doorOpen = false;
      System.out.println(“Door is closed.”);
   }

   public void goUp() {
      System.out.println(“Going up one floor.”);
      currentFloor++;
      System.out.println(“Floor: “ + currentFloor);
   }

   public void goDown() {
      System.out.println(“Going down one floor.”);
      currentFloor--;
      System.out.println(“Floor: “ + currentFloor);
   }

   public void setFloor(int desiredFloor) {
      while (currentFloor != desiredFloor){
      if (currentFloor < desiredFloor){
```

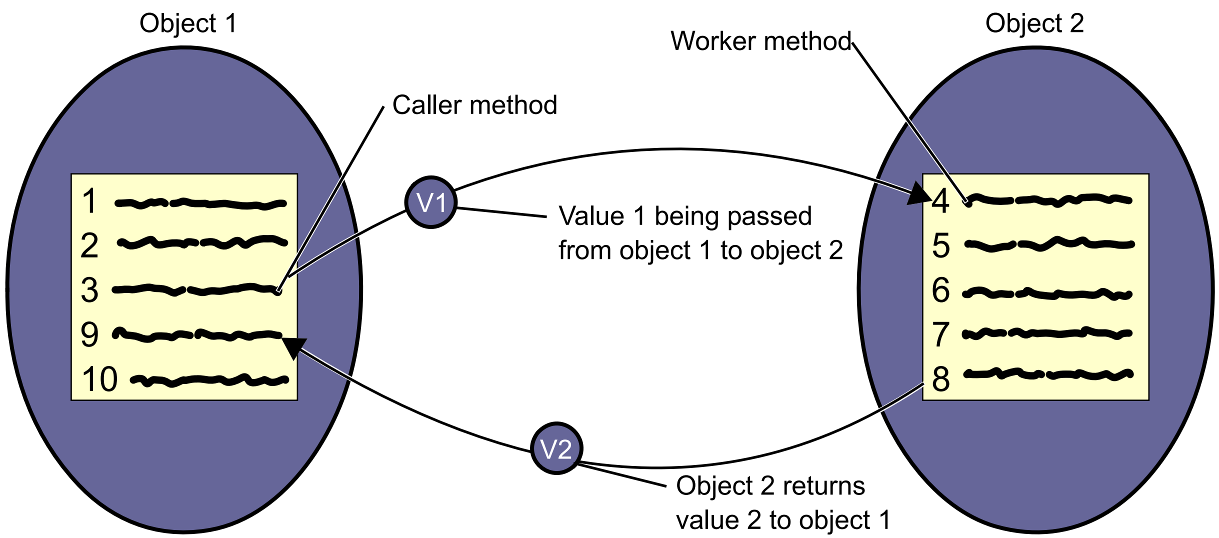
Invoking a Method in the Same Class

```
      goUp();
      }
    else {
      goDown();
      }
  }
  }

  public int getFloor() {
    return currentFloor;
  }

  public boolean checkDoorStatus() {
    return doorOpen;
  }
}

```

Guidelines for Invoking Methods

- There is no limit to the number of method calls that a calling method can make.
- The calling method and the worker method can be in the same class or in different classes.
- The way you invoke the worker method is different, depending on whether it is in the same class or in a different class from the calling method.
- You can invoke methods in any order. Methods do not need to be completed in the order in which they are listed in the class where they are declared (the class containing the worker methods).

Passing Arguments and Returning Values



Declaring Methods With Arguments

- Example:

```
public void setFloor(int desiredFloor) {
    while (currentFloor != desiredFloor) {
      if (currentFloor < desiredFloor) {
       goUp();
      }
      else {
       goDown();
      }
   }
}
```

- Example:

```
public void multiply(int numberOne, int numberTwo)
```

The `main` Method

- Example:

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

- Example (invocation):

```
java ShirtTest 12.99 R
```

Invoking Methods With Arguments

Example:

```
public class ElevatorTest {

   public static void main(String args[]) {

    Elevator myElevator = new Elevator();

     myElevator.openDoor();
     myElevator.closeDoor();
     myElevator.goUp();
     myElevator.goUp();
     myElevator.goUp();
     myElevator.openDoor();
     myElevator.closeDoor();
     myElevator.goDown();
     myElevator.openDoor();
     myElevator.closeDoor();
```

Invoking Methods With Arguments

```
        myElevator.goDown();

        myElevator.setFloor(myElevator.TOP_FLOOR);

        myElevator.openDoor();
    }
}

```

Declaring Methods With Return Values

Declaration:

```
public int sum(int numberOne, int numberTwo)
```

Returning a Value

- Example:

```
public int sum(int numberOne, int numberTwo) {
     int result= numberOne + numberTwo;

     return result;
}
```

- Example:

```
public int getFloor() {
    return currentFloor;
  }
```

Receiving Return Values

Example:

```
public class ElevatorTestTwo {

    public static void main(String args[]) {

     Elevator myElevator = new Elevator();

      myElevator.openDoor();
      myElevator.closeDoor();
      myElevator.goUp();
      myElevator.goUp();
      myElevator.goUp();
      myElevator.openDoor();
      myElevator.closeDoor();
      myElevator.goDown();
      myElevator.openDoor();
      myElevator.closeDoor();
```

Receiving Return Values

```
myElevator.goDown();

      int curFloor = myElevator.getFloor();
      System.out.println("Current Floor: " + curFloor);

      myElevator.setFloor(curFloor+1);

      myElevator.openDoor();
    }
}

```

Advantages of Method Use

- Methods make programs more readable and easier to maintain.
- Methods make development and maintenance quicker.
- Methods are central to reusable software.
- Methods allow separate objects to communicate and to distribute the work performed by the program.

Creating `static` Methods and Variables

- Comparing instance and `static` methods and variables
- Declaring `static` methods:

```
static Properties getProperties()
```

- Invoking `static` methods:

```
Classname.method();
```

Creating `static` Methods and Variables

- Example:

```
public static char convertShirtSize(int numericalSize) {

    if (numericalSize < 10) {
      return 'S';
    }

    else if (numericalSize < 14) {
      return 'M';
    }

    else if (numericalSize < 18) {
      return 'L';
    }

    else {
      return 'X';
   }
}
```

Creating `static` Methods and Variables

- Example:

```
char size = Shirt.convertShirtSize(16);
```

Creating `static` Methods and Variables

- Declaring `static` variables:

  ```
  static double salesTAX = 8.25;
  ```

- Accessing `static` variables:

  ```
  Classname.variable;
  ```

- Example:

  ```
  double myPI;
  myPI = Math.PI;
  ```

Static Methods and Variables in the Java API

Examples:

- The `Math` class
- The `System` class

Static Methods and Variables in the Java API

When to declare a `static` method or variable:

- Performing the operation on an individual object or associating the variable with a specific object type is not important.
- Accessing the variable or method before instantiating an object is important.
- The method or variable does not logically belong to an object, but possibly belongs to a utility class, such as the `Math` class, included in the Java API.

Using Method Overloading

Example overloaded methods:

```

public class Calculator {

    public int sum(int numberOne, int numberTwo){

      System.out.println("Method One");

      return numberOne + numberTwo;
    }

   public float sum(float numberOne, float numberTwo) {

      System.out.println("Method Two");

      return numberOne + numberTwo;
   }
```

Using Method Overloading

```
    public float sum(int numberOne, float numberTwo) {

      System.out.println("Method Three");

      return numberOne + numberTwo;
    }
}

```

Using Method Overloading

Example method invocation:

```
public class CalculatorTest {

  public static void main(String [] args) {

    Calculator myCalculator = new Calculator();

    int totalOne = myCalculator.sum(2,3);
    System.out.println(totalOne);

    float totalTwo = myCalculator.sum(15.99F, 12.85F);
    System.out.println(totalTwo);

    float totalThree = myCalculator.sum(2, 12.85F);
    System.out.println(totalThree);
  }
}
```

Method Overloading and the Java API

Method	**Use**
`void println()`	Terminates the current line by writing the line separator string
`void println(boolean x)`	Prints a `boolean value` and then terminates the line
`void println(char x)`	Prints a character and then terminates the line
`void println(char[] x)`	Prints an array of characters and then terminates the line
`void println(double x)`	Prints a `double` and then terminates the line
`void println(float x)`	Prints a `float` and then terminates the line
`void println(int x)`	Prints an `int` and then terminates the line
`void println(long x)`	Prints a `long` and then terminates the line
`void println(Object x)`	Prints an object and then terminates the line
`void println(String x)`	Prints a string and then terminates the line

Uses for Method Overloading

Examples:

```
public int sum(int numberOne, int numberTwo)
public int sum(int numberOne, int numberTwo, int numberThree)
public int sum(int numberOne, int numberTwo,int numberThree, int
numberFour)
```

Uses for Method Overloading

Example:

```
public class ShirtTwo {

  public int shirtID = 0; // Default ID for the shirt
  public String description = “-description required-”; // default

  // The color codes are R=Red, B=Blue, G=Green, U=Unset
  public char colorCode = ‘U’;
  public double price = 0.0; // Default price for all items
  public int quantityInStock = 0; // Default quantity for all items

  public void setShirtInfo(int ID, String desc, double cost){
    shirtID = ID;
    description = desc;
    price = cost;
  }

```

Uses for Method Overloading

```
public void setShirtInfo(int ID, String desc, double cost, char color){
    shirtID = ID;
    description = desc;
    price = cost;
    colorCode = color;
  }

  public void setShirtInfo(int ID,  String desc, double cost, char color, int quantity){
    shirtID = ID;
    description = desc;
    price = cost;
    colorCode = color;
    quantityInStock = quantity;
  }

  // This method displays the values for an item
  public void display() {

```

Uses for Method Overloading

```
      System.out.println(“Item ID: “ + shirtID);
      System.out.println(“Item description:” + description);
      System.out.println(“Color Code: “ + colorCode);
      System.out.println(“Item price: “ + price);
      System.out.println(“Quantity in stock: “ + quantityInStock);

   } // end of display method
} // end of class

```

Uses for Method Overloading

Example:

```
class ShirtTwoTest {

  public static void main (String args[]) {
    ShirtTwo shirtOne = new ShirtTwo();
    ShirtTwo shirtTwo = new ShirtTwo();
    ShirtTwo shirtThree = new ShirtTwo();

    shirtOne.setShirtInfo(100, "Button Down", 12.99);
    shirtTwo.setShirtInfo(101, "Long Sleeve Oxford", 27.99, 'G');
   shirtThree.setShirtInfo(102, "Shirt Sleeve T-Shirt", 9.99, 'B', 50);

    shirtOne.display();
    shirtTwo.display();
    shirtThree.display();
  }
}

```

Module 9

Implementing Encapsulation and Constructors

Objectives

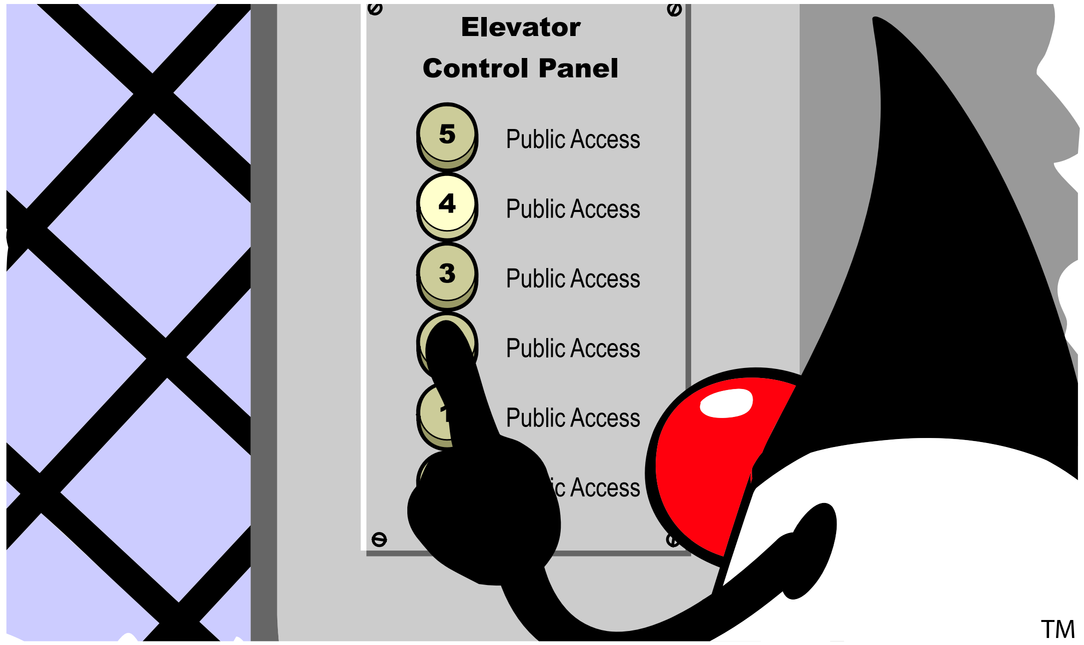
- Use encapsulation to protect data
- Create constructors to initialize objects

Relevance

- The earliest elevators, or lifts, required a user to manipulate one or more pulleys, ropes, and gears to operate the elevator. Modern elevators hide the detail and only require a user to push a few buttons to operate an elevator. What are the advantages of modern elevators over the older elevators?
- Many elevators, such as a service elevator in a factory, require keys before they can be operated. Other elevators require a key to travel to a particular floor, such as the top floor in a hotel. Why are these keys important?
- What do you think of when you hear the words *private* and *public*?

Using Encapsulation

The `public` Modifier




```
public int currentFloor=1;

public void setFloor(int desiredFloor) {
      ...
}
```

The `public` Modifier

Example:

```
public class PublicElevator {

    public boolean doorOpen=false;
    public int currentFloor = 1;
    public int weight =0;

    public final int CAPACITY=1000;
    public final int TOP_FLOOR = 5;
    public final int BOTTOM_FLOOR = 1;
}

```

The `public` Modifier

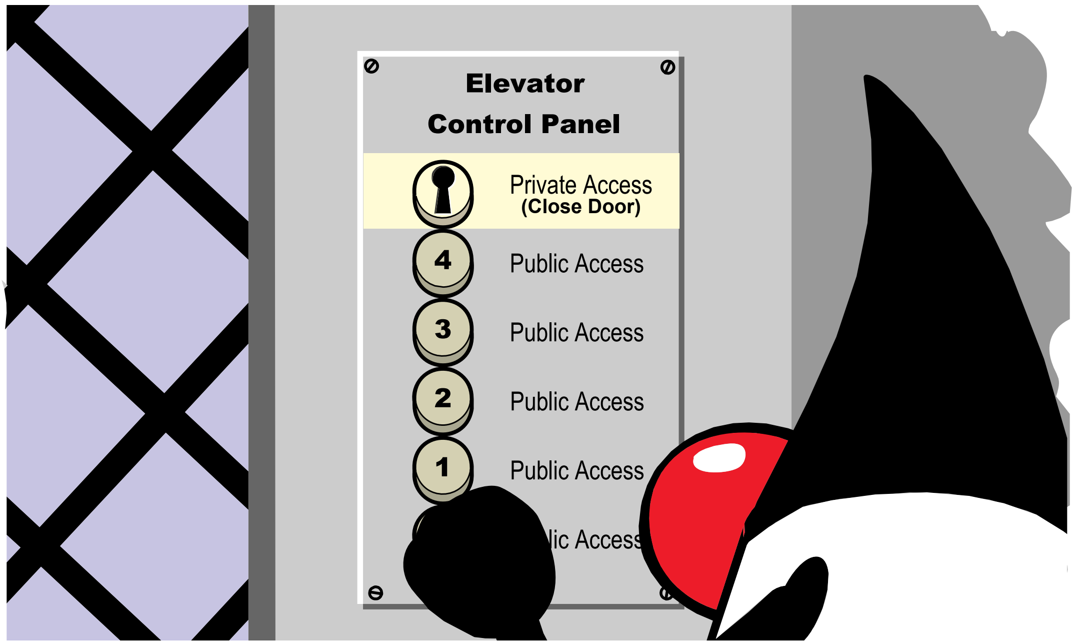
Example:

```

public class PublicElevatorTest {

  public static void main(String args[]) {

    PublicElevator pubElevator = new PublicElevator();

    pubElevator.doorOpen = true;  //passengers get on
    pubElevator.doorOpen = false; //doors close
    //go down to floor 0 (below bottom of building)
    pubElevator.currentFloor--;
    pubElevator.currentFloor++;

    //jump to floor 7 (only 5 floors in building)
    pubElevator.currentFloor = 7;
    pubElevator.doorOpen = true;  //passengers get on/off
    pubElevator.doorOpen = false;
```

The `public` Modifier

```
pubElevator.currentFloor = 1; //go to the first floor
    pubElevator.doorOpen = true;  //passengers get on/off
    pubElevator.currentFloor++;   //elevator moves with door open
    pubElevator.doorOpen = false;
    pubElevator.currentFloor--;
    pubElevator.currentFloor--;
  }
}
```

The `private` Modifier




```
private int currentFloor=1;
private void calculateCapacity() {
    ...
}
```

The `private` Modifier

Example:

```

public class PrivateElevator1 {

   private boolean doorOpen=false;
   private int currentFloor = 1;
   private int weight =0;

   private final int CAPACITY=1000;
   private final int TOP_FLOOR = 5;
   private final int BOTTOM_FLOOR = 1;
}

```

The `private` Modifier

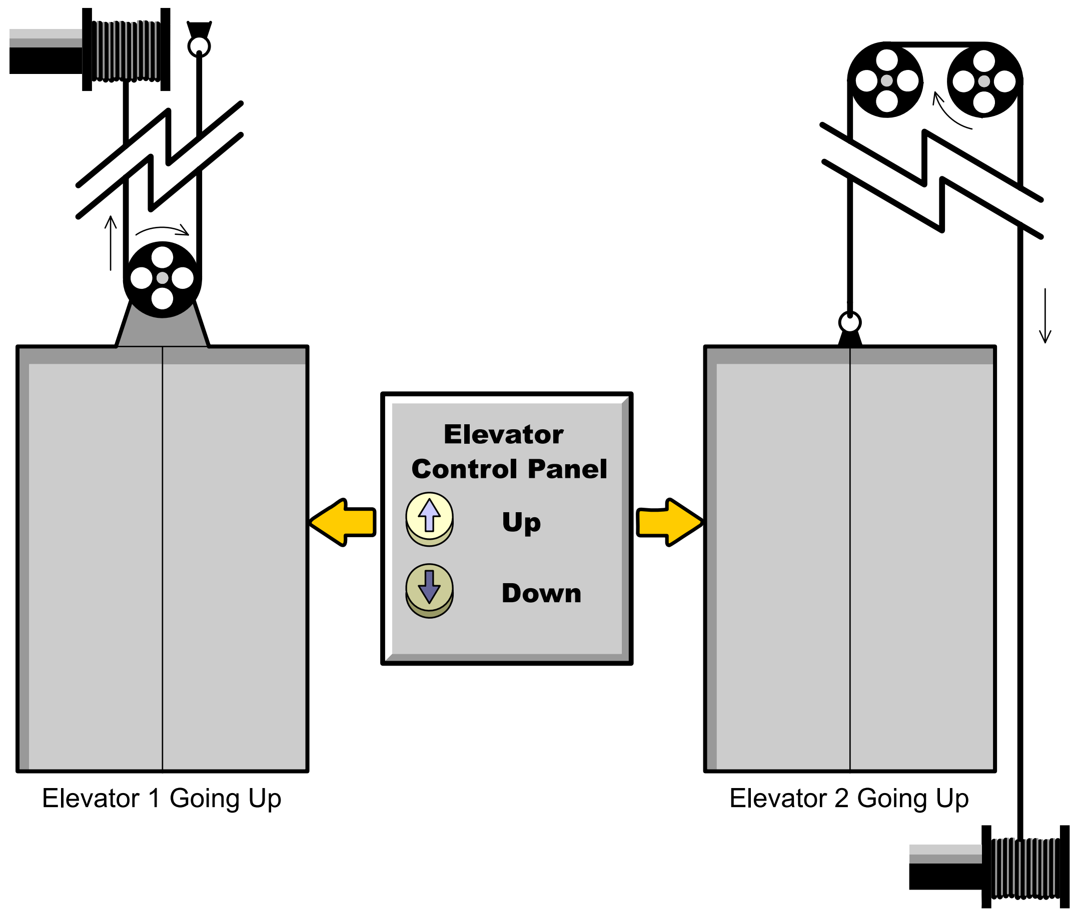
Example:

```

public class PrivateElevator1Test {

  public static void main(String args[]) {

    PrivateElevator1 privElevator = new PrivateElevator1();

    /*************************************************
    * The following lines of code will not compile    *
    * because they attempt to access private          *
    * variables.                                      *
    *************************************************/

    privElevator.doorOpen = true;  //passengers get on
    privElevator.doorOpen = false; //doors close
    //go down to currentFloor 0 (below bottom of building)
    privElevator.currentFloor--;
```

The `private` Modifier

```
        privElevator.currentFloor++;

        //jump to currentFloor 7 (only 5 floors in building)
        privElevator.currentFloor = 7;
        privElevator.doorOpen = true;  //passengers get on/off
        privElevator.doorOpen = false;
        privElevator.currentFloor = 1; //go to the first floor
        privElevator.doorOpen = true;  //passengers get on/off
        privElevator.currentFloor++;   //elevator moves with door open
        privElevator.doorOpen = false;
        privElevator.currentFloor--;
        privElevator.currentFloor--;
    }
}
```

Interface and Implementation



Interface and Implementation

Example:

```

public class PrivateShirt1 {

  private int shirtID = 0; // Default ID for the shirt
  private String description = “-description required-”; // default

  // The color codes are R=Red, B=Blue, G=Green, U=Unset
  private char colorCode = ‘U’;
  private double price = 0.0; // Default price for all items
  private int quantityInStock = 0; // Default quantity for all items

  public char getColorCode() {
    return colorCode;
  }

  public void setColorCode(char newCode) {
    colorCode = newCode;
```

Interface and Implementation

```
}

  // Additional get and set methods for shirtID, description,
  // price, and quantityInStock would follow

} // end of class

```

Interface and Implementation

Example:

```java

public class PrivateShirt1Test {

  public static void main (String args[]) {

  PrivateShirt1 privShirt = new PrivateShirt1();
  char colorCode;

  // Set a valid colorCode
  privShirt.setColorCode(‘R’);
  colorCode = privShirt.getColorCode();

  // The PrivateShirtTest1 class can set a valid colorCode
  System.out.println(“Color Code: “ + colorCode);

  // Set an invalid color code
  privShirt.setColorCode(‘Z’);
```

Interface and Implementation

```
colorCode = privShirt.getColorCode();

  // The PrivateShirtTest1 class can set an invalid colorCode
  System.out.println("Color Code: " + colorCode);
  }
}

```

Interface and Implementation

Example:

```

public class PrivateShirt2 {

  private int shirtID = 0; // Default ID for the shirt
  private String description = “-description required-”; // default

  // The color codes are R=Red, B=Blue, G=Green, U=Unset
  private char colorCode = ‘U’;
  private double price = 0.0; // Default price for all items
  private int quantityInStock = 0; // Default quantity for all items

  public char getColorCode() {
    return colorCode;
  }

  public void setColorCode(char newCode) {

```

Interface and Implementation

```
switch (newCode) {
    case 'R':
    case 'G':
    case 'B':
      colorCode = newCode;
      break;
    default:
      System.out.println("Invalid colorCode. Use R, G, or B");
    }
  }

  // Additional get and set methods for shirtID, description,
  // price, and quantityInStock would follow

} // end of class

```

Interface and Implementation

Example:

```
public class PrivateShirt2Test {

  public static void main (String args[]) {
    PrivateShirt2 privShirt = new PrivateShirt2();
    char colorCode;

    // Set a valid colorCode
    privShirt.setColorCode(‘R’);
    colorCode = privShirt.getColorCode();

    // The PrivateShirtTest2 class can set a valid colorCode
    System.out.println(“Color Code: “ + colorCode);

    // Set an invalid color code
    privShirt.setColorCode(‘Z’);
    colorCode = privShirt.getColorCode();
```

Interface and Implementation

```

        // The PrivateShirtTest2 class cannot set an invalid colorCode.
        // Color code is still R
        System.out.println("Color Code: " + colorCode);
    }
}

```

Encapsulated Elevator

Example:

```

public class PrivateElevator2 {

  private boolean doorOpen=false;
  private int currentFloor = 1;
  private int weight = 0;

  private final int CAPACITY = 1000;
  private final int TOP_FLOOR = 5;
  private final int BOTTOM_FLOOR = 1;

  public void openDoor() {
    doorOpen = true;
  }

  public void closeDoor() {
    calculateCapacity();
```

Encapsulated Elevator

```

    if (weight <= CAPACITY) {
      doorOpen = false;
    }
    else {
      System.out.println("The elevator has exceeded capacity.");
     System.out.println("Doors will remain open until someone exits!");
    }
  }

  // In reality, the elevator would have weight sensors to
  // check the actual weight in the elevator, but for the sake
  // of simplicity we just pick a random number to represent the
  // weight in the elevator

   private void calculateCapacity() {
    weight = (int) (Math.random() * 1500);
    System.out.println("The weight is " + weight);
  }
```

Encapsulated Elevator

```

  public void goUp() {
    if (!doorOpen) {
      if (currentFloor < TOP_FLOOR) {
      currentFloor++;
      System.out.println(currentFloor);
      }
      else {
        System.out.println("Already on top floor.");
        }
    }
    else {
      System.out.println("Doors still open!");
      }
    }

  public void goDown() {
    if (!doorOpen) {
    if (currentFloor > BOTTOM_FLOOR) {
```

Encapsulated Elevator

```
            currentFloor--;
            System.out.println(currentFloor);
            }
          else {
            System.out.println("Already on bottom floor.");
            }
          }
          else {
            System.out.println("Doors still open!");
            }
        }

        public void setFloor(int desiredFloor) {
          if ((desiredFloor >= BOTTOM_FLOOR) && (desiredFloor<=TOP_FLOOR)) {

            while (currentFloor != desiredFloor) {
        if (currentFloor < desiredFloor) {
          goUp();
        }
```

Encapsulated Elevator

```

    else {
      goDown();
    }
        }
      }
      else {
        System.out.println("Invalid Floor");
      }
    }

    public int getFloor() {
      return currentFloor;
    }

    public boolean getDoorStatus() {
      return doorOpen;
    }
}
```

Encapsulated Elevator

Example:

```

public class PrivateElevator2Test {

   public static void main(String args[]) {

     PrivateElevator2 privElevator = new PrivateElevator2();

     privElevator.openDoor();
     privElevator.closeDoor();
     privElevator.goDown();
     privElevator.goUp();
     privElevator.goUp();
     privElevator.openDoor();
     privElevator.closeDoor();
     privElevator.goDown();
     privElevator.openDoor();
     privElevator.goDown();
```

Encapsulated Elevator

```
privElevator.closeDoor();
        privElevator.goDown();
        privElevator.goDown();

        int curFloor = privElevator.getFloor();

        if (curFloor != 5 && ! privElevator.getDoorStatus()) {
            privElevator.setFloor(5);
        }

        privElevator.setFloor(10);
        privElevator.openDoor();
    }
}

```

Sample Output

```
The weight is 453
Already on bottom floor.
2
3
The weight is 899
2
Doors still open!
The weight is 974
1
Already on bottom floor.
2
3
4
5
```

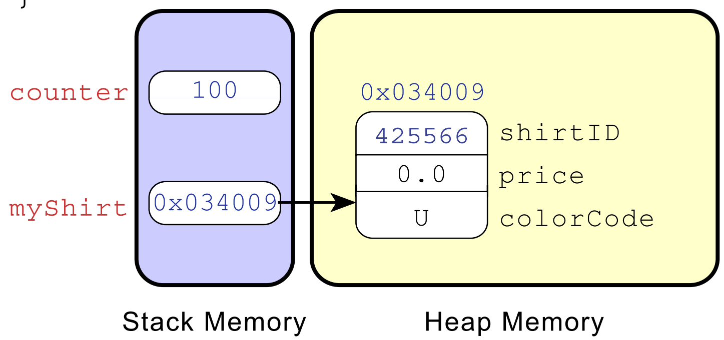
Describing Variable Scope

Example:

```
public class Person2 {

  // begin scope of int age
  private int age = 34;

  public void displayName() {
    // begin scope of String name
    String name = "Peter Simmons";
    System.out.println("My name is " + name + " and I am " + age );

  }   // end scope of String name

  public String getName () {

    return name; // this causes an error
  }
}   // end scope of int age
```

How Instance Variables and Local Variables Appear in Memory

```
public static void main (String args[]) {

  int counter = 100;
  Shirt myShirt = new Shirt ( );
  myShirt.shirtID = 425566 ;
}
```



Creating Constructors

Syntax:

```
[modifiers] class ClassName {

     [modifiers] ConstructorName([arguments]) {
       code_block
     }
}
```

Creating Constructors

Example:

```

public class ConstructorShirt1 {

  private int shirtID = 0; // Default ID for the shirt
  private String description = “-description required-”; // default

  // The color codes are R=Red, B=Blue, G=Green, U=Unset
  private char colorCode = ‘U’;
  private double price = 0.0; // Default price for all items
  private int quantityInStock = 0; // Default quantity for all items

  public ConstructorShirt1(char startingCode) {

    switch (startingCode) {
    case ‘R’:
    case ‘G’:
    case ‘B’:
```

Creating Constructors

```
colorCode = startingCode;
      break;
    default:
      System.out.println("Invalid colorCode. Use R, G, or B");
    }
  }

  public char getColorCode() {
    return colorCode;
  }
} // end of class

```

Creating Constructors

Example:

```
public class ConstructorShirt1Test {

  public static void main (String args[]) {

    ConstructorShirt1 constShirt = new ConstructorShirt1(‘R’);
    char colorCode;

    colorCode = constShirt.getColorCode();

    System.out.println(“Color Code: “ + colorCode);
  }
}
```

Default Constructor

- Example:

```
ConstructorShirt1 constShirt = new ConstructorShirt1();
```

- Example:

```

public class DefaultShirt {

  private int shirtID = 0; // Default ID for the shirt
  private String description = “-description required-”; // default

  // The color codes are R=Red, B=Blue, G=Green, U=Unset
  private char colorCode = ‘U’;
  private double price = 0.0; // Default price for all items
  private int quantityInStock = 0; // Default quantity for all items

  public DefaultShirt() {
    colorCode = ‘R’;
  }
```

Default Constructor

```

  public char getColorCode() {
    return colorCode;
  }
} // end of class

```

Overloading Constructors

Example:

```
public class ConstructorShirt2 {

  private int shirtID = 0; // Default ID for the shirt
  private String description = "-description required-"; // default

  // The color codes are R=Red, B=Blue, G=Green, U=Unset
  private char colorCode = 'U';
  private double price = 0.0; // Default price for all items
  private int quantityInStock = 0; // Default quantity for all items

  public ConstructorShirt2() {
    colorCode = 'R';
  }

  public ConstructorShirt2 (char startingCode) {

```

Overloading Constructors

```
switch (startingCode) {
    case ‘R’:
    case ‘G’:
    case ‘B’:
      colorCode = startingCode;
      break;
    default:
      System.out.println(“Invalid colorCode. Use R, G, or B”);
    }
  }
  public ConstructorShirt2 (char startingCode, int startingQuantity) {

    switch (startingCode) {
    case ‘R’:
      colorCode = startingCode;
      break;
    case ‘G’:
      colorCode = startingCode;
      break;
```

Overloading Constructors

```
case ‘B’:
      colorCode = startingCode;
      break;
    default:
      System.out.println(“Invalid colorCode. Use R, G, or B”);
    }

    if (startingQuantity > 0 && startingQuantity < 2000) {
      quantityInStock = startingQuantity;
    }

    else {
      System.out.println(“Invalid quantity. Must be > 0 or < 2000”);
    }
  }

  public char getColorCode() {
    return colorCode;
    }
```

Overloading Constructors

```
public int getQuantityInStock() {
        return quantityInStock;
  }

} // end of class

```

Overloading Constructors

Example:

```

public class ConstructorShirt2Test {

  public static void main (String args[]) {

    ConstructorShirt2 constShirtFirst = new ConstructorShirt2();
    ConstructorShirt2 constShirtSecond = new ConstructorShirt2('G');
    ConstructorShirt2 constShirtThird = new ConstructorShirt2('B', 1000);

    char colorCode;
    int quantity;

    colorCode = constShirtFirst.getColorCode();
    System.out.println("Object 1 Color Code: " + colorCode);

    colorCode = constShirtSecond.getColorCode();
```

Overloading Constructors

```
System.out.println(“Object 2 Color Code: “ + colorCode);

    colorCode = constShirtThird.getColorCode();
    quantity = constShirtThird.getQuantityInStock();
    System.out.println(“Object 3 Color Code: “ + colorCode);
    System.out.println(“Object 3 Quantity on Hand: “ + quantity);
  }
}

```

Module 10

Creating and Using Arrays

Objectives

- Code one-dimensional arrays
- Set array values using the `length` attribute and a loop
- Pass arguments to the `main` method for use in a program
- Create two-dimensional arrays

Relevance

- An array is an orderly arrangement of something, such as an ordered list. What are some things that people use arrays for in their daily lives?
- If a one-dimensional array is a list of items, what is a two-dimensional array?
- How do you access items in an array?

Creating One-Dimensional Arrays

Example:

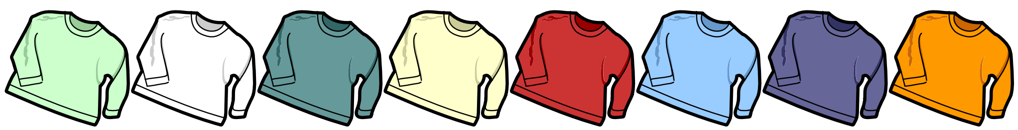
```
int ageOne = 27;
int ageTwo = 12;
int ageThree = 82;
int ageFour = 70;
int ageFive = 54;
int ageSix = 6;
int ageSeven = 1;
int ageEight = 30;
int ageNine = 34;
int ageTen = 42;
```

Creating One-Dimensional Arrays

Array of `int`

425566	15	200	1	1151	7205	8000	609834

Array of `Shirts`



Array of `Strings`

`Hugh Mongus` `Aaron Datires` `Stan Ding` `Albert Kerkie` `Carrie DeKeys` `Walter Mellon` `Hugh Morris` `Moe DeLawn`

Declaring a One-Dimensional Array

- Syntax:

```
type [] array_identifier;
```

- Examples:

```
char [] status;
int [] ages;

Shirt [] shirts;
String [] names;
```

Instantiating a One-Dimensional Array

- Syntax:

  array_identifier = new type [length];

- Examples:

```
status = new char [20];
ages = new int [5];

names = new String [7];
shirts = new Shirt [3];
```

Initializing a One-Dimensional Array

- Syntax:

  array_identifier[index] = value;

- Examples:

```
ages[0] = 19;
ages[1] = 42;
ages[2] = 92;
ages[3] = 33;
ages[4] = 46;

shirts[0] = new Shirt();
shirts[1] = new Shirt('G');
shirts[2] = new Shirt('G', 1000);
```

Declaring, Instantiating, and Initializing One-Dimensional Arrays

- Syntax:

```
type [] array_identifier = {comma-separated list of values or expressions};
```

- Examples:

```
int [] ages = {19, 42, 92, 33, 46};

Shirt [] shirts = {new Shirt(), new Shirt('G'), new Shirt('g',1000)};
```

- Error:

```
int [] ages;
ages = {19, 42, 92, 33, 46};
```

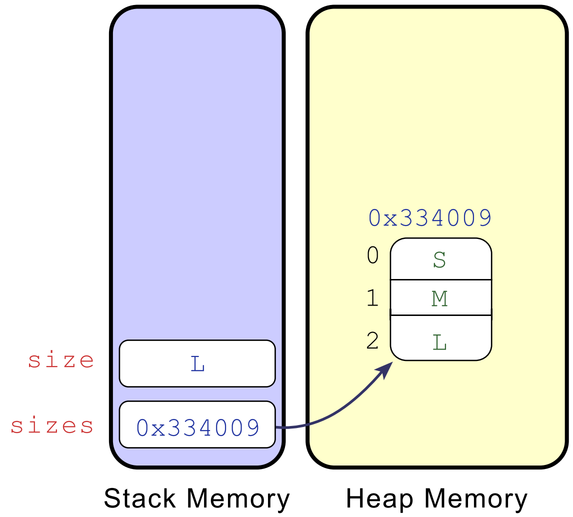
Accessing a Value Within an Array

Examples:

```
status[0] = '3';
names[1] = "Fred Smith";
ages[1] = 19;
prices[2] = 9.99F;

char s = status[0];
String name = names [1];
int age = ages[1];
double price = prices[2];
```

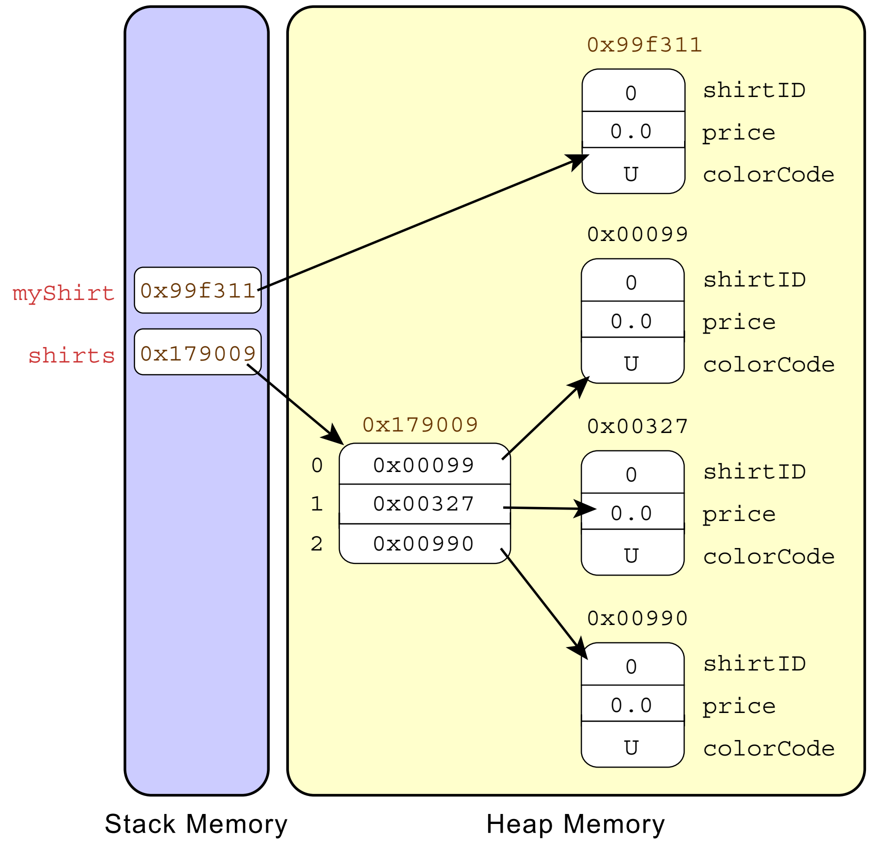
Storing Primitive Variables and Arrays of Primitives in Memory

```
char size = 'L'
char [] sizes = {'S','M','L'};
```



Storing Arrays of References in Memory

```
1 Shirt myShirt = new Shirt();
2 Shirt [] shirts = {new Shirt(),
                     new Shirt(),
                     new Shirt()};
```



Setting Array Values Using the `length` Attribute and a Loop

Example:

```
int [] myArray;
myArray = new int[100];

for (int count = 0; count < myArray.length; count++) {
     myArray[count] = count;
}
```

Enhanced For Loop

- The enhanced for loop can be used to make your loops more compact and easy to read
- This form of the for loop is designed for iteration through arrays
- Example:

```
class ExampleFor {
     public static void main(String[] args){
          int[] numbers = {1,3,5,7,9,11,13,15,17,19};
          int sum=0;
          for (int item : numbers) {
           sum = sum + item;
          }
          System.out.println("Sum is: " + sum);

     }
}
```

Using the `args` Array in the `main` Method

- Example:

```
public static void main (String args[]);
```

- Example:

```
public class ArgsTest {

  public static void main (String args[]) {

    System.out.println("args[0] is " + args[0]);
    System.out.println("args[1] is " + args[1]);
  }
}

```

Converting `String` Arguments to Other Types

Example:

```
int ID = Integer.parseInt(args[0]);
```

The Varargs Feature

- You can create a method that can accept variable number of arguments.
- A method can have at most one parameter that is a vararg
- Vararg must be the last parameter taken by the method. It is denoted by an object type, a set of ellipses (...), and the name of the variable. For example:

```
class VarMessage{
   public static void showMessage(String... names) {
     for (String list: names)
       System.out.println(list);
   }
   public static void main (String args[]){
     showMessage (args)
}
```

Describing Two-Dimensional Arrays

	Sunday	Monday	Tuesday	Wedsday	Thursday	Friday	Saturday
Week 1							
Week 2							
Week 3							
Week 4							

Declaring a Two-Dimensional Array

- Syntax:

  `type [][] array_identifier;`

- Example:

```
int [][] yearlySales;
```

Instantiating a Two-Dimensional Array

- Syntax:

```
array_identifier = new type [number_of_arrays] [length];
```

- Example:

```
// Instantiates a two-dimensional array: 5 arrays of 4 elements each
yearlySales = new int[5][4];
```

	Quarter 1	Quarter 2	Quarter 3	Quarter 4
Year 1				
Year 2				
Year 3				
Year 4				
Year 5				

Initializing a Two-Dimensional Array

Example:

```
yearlySales[0][0] = 1000;
yearlySales[0][1] = 1500;
yearlySales[0][2] = 1800;
yearlySales[1][0] = 1000;
yearlySales[2][0] = 1400;
yearlySales[3][3] = 2000;
```

	Quarter 1	**Quarter 2**	**Quarter 3**	**Quarter 4**
Year 1	1000	1500	1800	
Year 2	1000			
Year 3	1400			
Year 4				2000
Year 5				

Module 11

Implementing Inheritance

Objectives

- Define and test your use of inheritance
- Explain abstraction
- Explicitly identify class libraries used in your code

Relevance

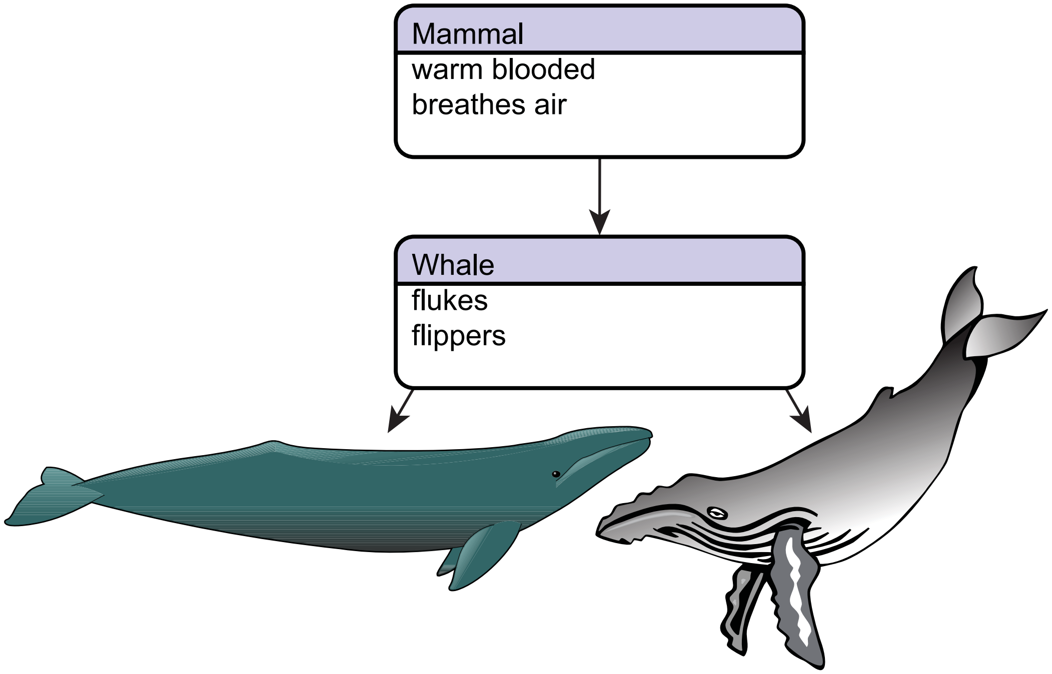
- Inheritance refers to the passing down of something from one organism to some another organism. What are some physical characteristics that you have inherited?
- From whom did you inherit your characteristics?
- What class hierarchy are you from?
- Did you inherit characteristics from multiple classes?
- What does it mean if something is “abstract?”
- What do you think an abstract class is?

Inheritance

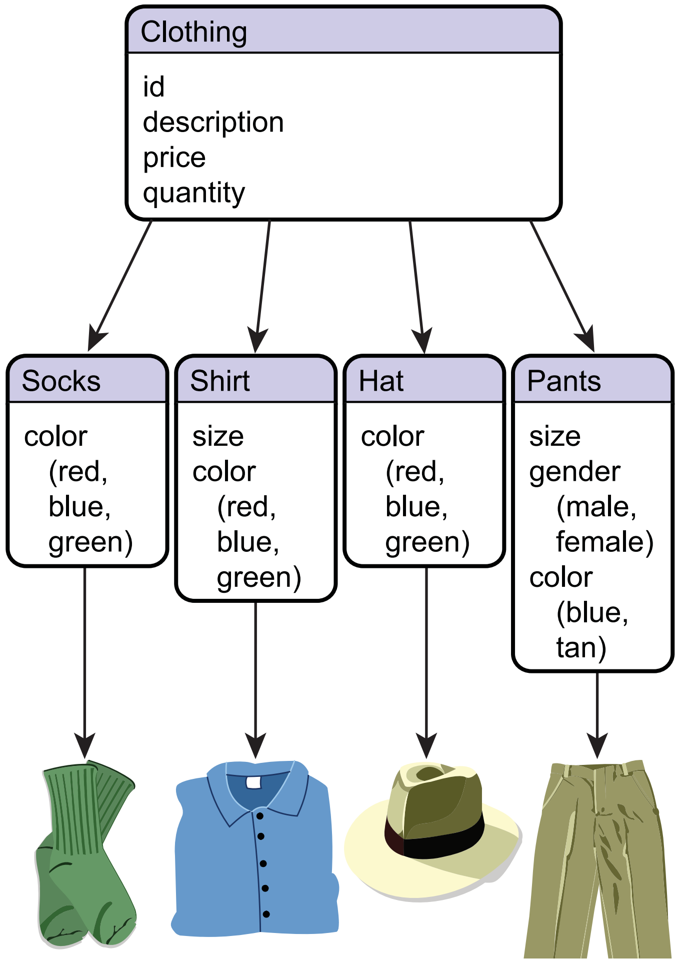
Hat	Sock
ID price description colorCode R=Red, B=Blue, G=Green quantityInStock	ID price description colorCode R=Red, B=Blue, G=Green quantityInStock
calculateID() displayInformation()	calculateID() displayInformation()

Pant	**Shirt**
ID price size gender M=Male, F=Female description colorCode B=Blue, T=Tan quantityInStock	ID price description colorCode R=Red, B=Blue, G=Green quantityInStock
calculateID() displayInformation()	calculateID() displayInformation()

Superclasses and Subclasses



Testing Superclass and Subclass Relationships



Modeling Superclasses and Subclasses

Hat:Clothing	Socks:Clothing
colorCode R=Red, B=Blue, G=Green	colorCode R=Red, B=Blue, G=Green
displayInformation()	displayInformation()

Pants:Clothing	Shirt:Clothing
size gender M=Male, F=Female colorCode B=Blue, T=Tan	size colorCode R=Red, B=Blue, G=Green
displayInformation()	displayInformation()

Modeling Superclasses and Subclasses

Clothing
ID price description quantityInStock
calculateID()

Declaring a Superclass

Example:

```

public class Clothing {

  private int ID = 0; // Default ID for all clothing
  private String description = "-description required-"; // default

  private double price = 0.0; // Default price for all clothing
  private int quantityInStock = 0; // Default quantity for all clothing

  private static int UNIQUE_ID=0; //Static member incremented in
constructor to generate uniqueId

  public Clothing() {
    ID = UNIQUE_ID++;
  }

public int getID() {
```

Declaring a Superclass

```
return ID;
}

  public void setDescription(String d) {
    description = d;
  }

  public String getDescription() {
    return description;
  }

  public void setPrice(double p) {
    price = p;
  }

  public double getPrice() {
    return price;
  }

```

Declaring a Superclass

```
public void setQuantityInStock(int q) {
quantityInStock = q;
  }

  public int getQuantityInStock() {
    return quantityInStock;
  }

} // end of class

```

Declaring a Subclass

Syntax:

```
[class_modifier] class class_identifier extends superclass_identifier
```

Declaring a Subclass

Example:

```java
public class Shirt extends Clothing {

  // The color codes are R=Red, B=Blue, G=Green, U=Unset
  public char colorCode = 'U';

  // This method displays the values for an item
  public void displayInformation() {

    System.out.println("Shirt ID: " + getID());
    System.out.println("Shirt description:" + getDescription());
    System.out.println("Color Code: " + colorCode);
    System.out.println("Shirt price: " + getPrice());
    System.out.println("Quantity in stock: " + getQuantityInStock());

  } // end of display method
} // end of class

```

Abstraction

- What is abstraction?
- Abstraction in the DirectClothing, Inc. case study

Classes in the Java API

- Implicitly available classes: the `java.lang` package
- Importing and qualifying classes:
  - The `java.awt` package
  - The `java.applet` package
  - The `java.net` package
  - The `java.io` package
  - The `java.util` package

The `import` Statement

- Syntax:

```
import package_name.class_name;
import package_name.*;
```

- Example:

```
import java.awt.*;
public class MyPushButton1 extends Button {
    // class statements

}
```

Specifying the Fully Qualified Name

- Syntax:

```
package_name.class_name
```

- Example:

```
public class MyPushButton2 extends java.awt.Button {
    // class statements
}
```