Fundamentals of the Java™ Programming Language

SL-110

.THIS MANUAL IS DESIGNED TO SUPPORT AN INSTRUCTOR-LED TRAINING (ILT) COURSE AND IS INTENDED TO BE USED FOR REFERENCE PURPOSES IN CONJUNCTION WITH THE ILT COURSE. THE MANUAL IS NOT A STANDALONE TRAINING TOOL. USE OF THE MANUAL FOR SELF-STUDY WITHOUT CLASS ATTENDANCE IS NOT RECOMMENDED.

CE MANUEL DE RÉFÉRENCE DOIT ÊTRE UTILISÉ DANS LE CADRE D'UN COURS DE FORMATION DIRIGÉ PAR UN INSTRUCTEUR (ILT). IL NE S'AGIT PAS D'UN OUTIL DE FORMATION INDÉPENDANT. NOUS VOUS DÉCONSEILLONS DE L'UTILISER DANS LE CADRE D'UNE AUTO-FORMATION.

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?

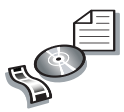
To be sure you are prepared to take this course, can you answer yes to the following questions?

- Can you create programs using a procedural language such as C or a scripting language such as Perl?
- 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

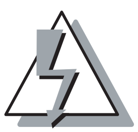
How to Use the Icons



Demonstration

Discussion

Note



Caution - Electrical

Caution - Heat

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.
- ***`Courier italics bold`*** is used to represent variables whose values are to be entered by the student as part of an activity.

Typographical Conventions

- *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 Environment (Solaris OE) is different from the Microsoft Windows platform, both commands are shown.

Module 1

Explaining Java™ Technology

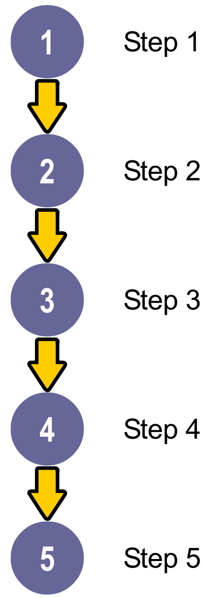
Overview

- 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

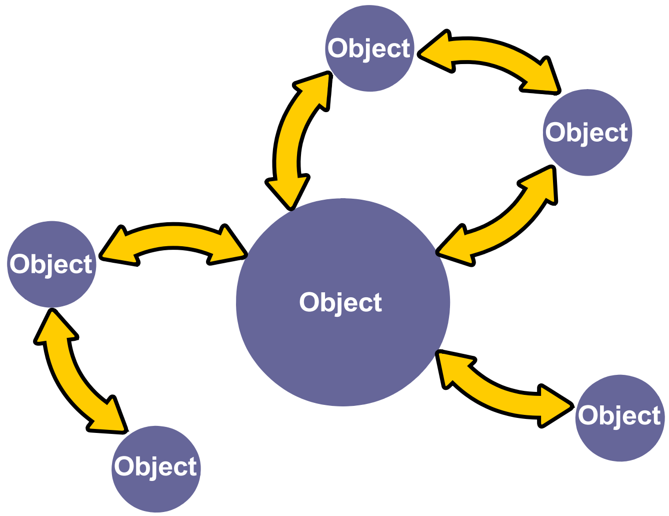
Key Concepts of the Java Programming Language

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

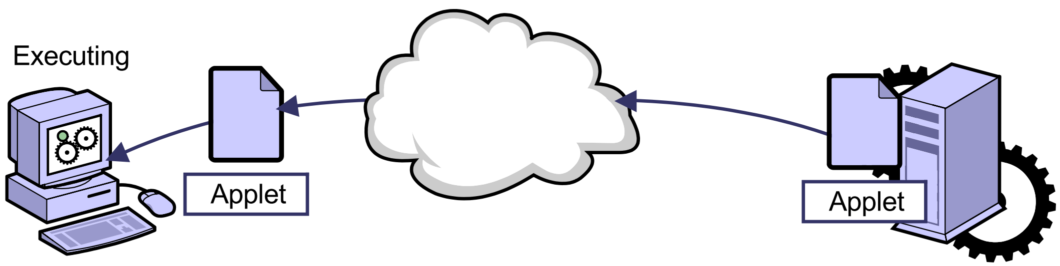
Object-Oriented



Object-Oriented



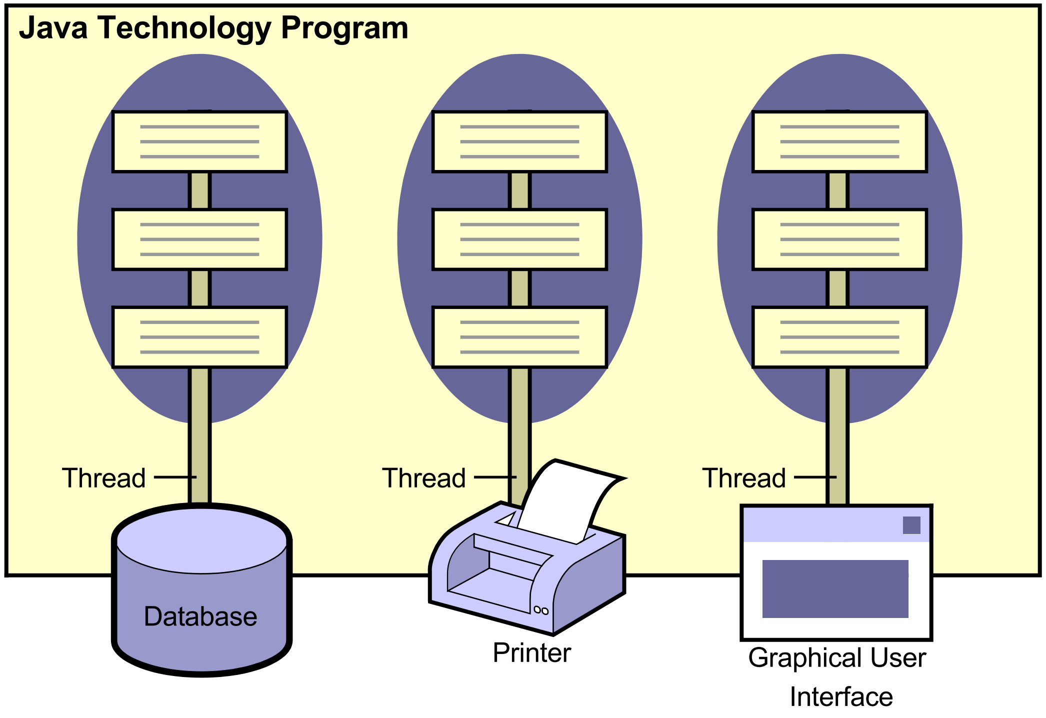
Distributed



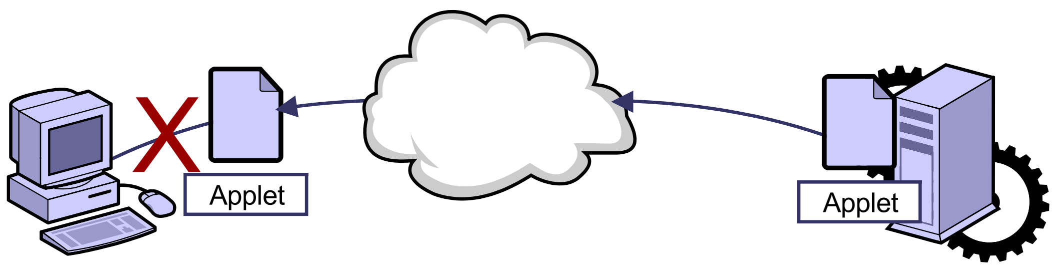
Simple

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

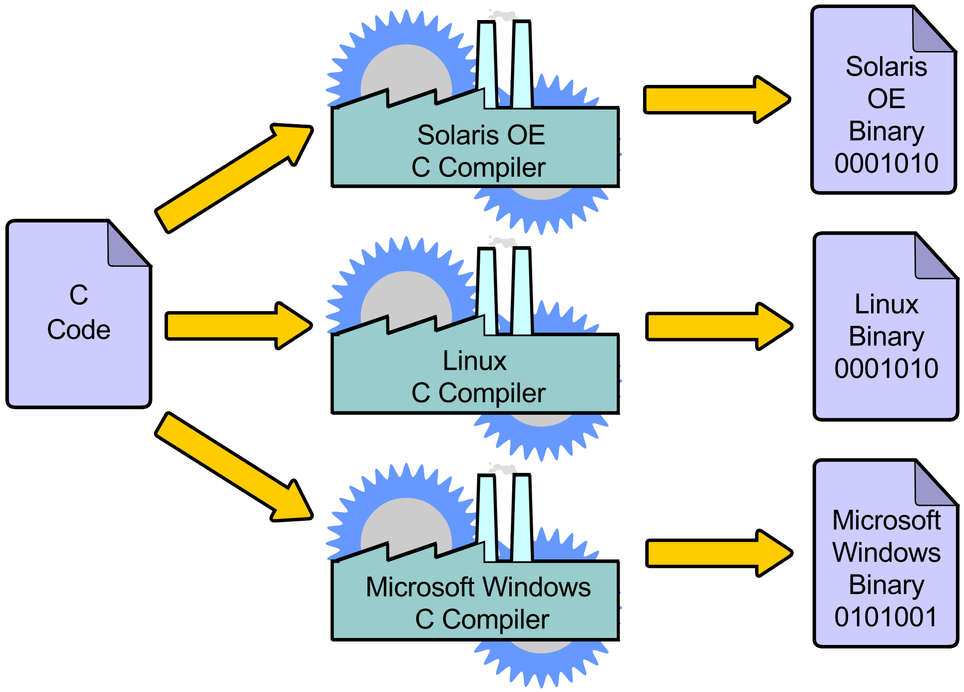
Multithreaded



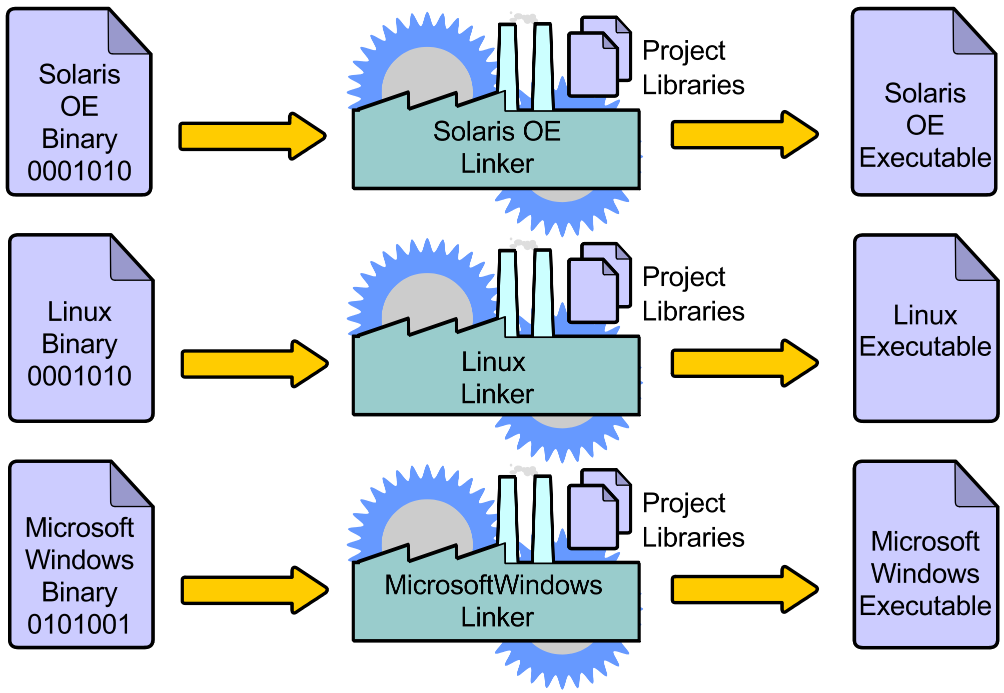
Secure



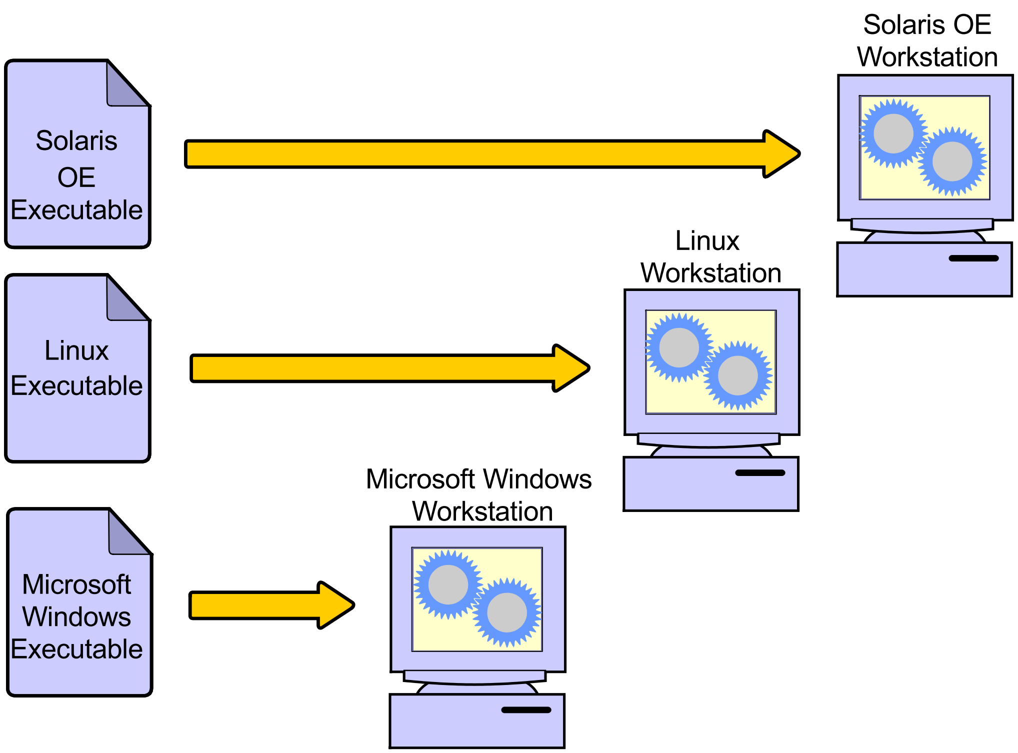
Platform-Dependent Programs



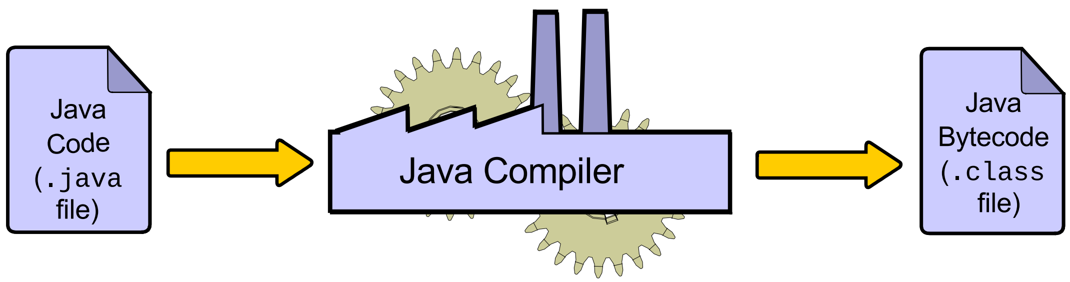
Platform-Dependent Programs



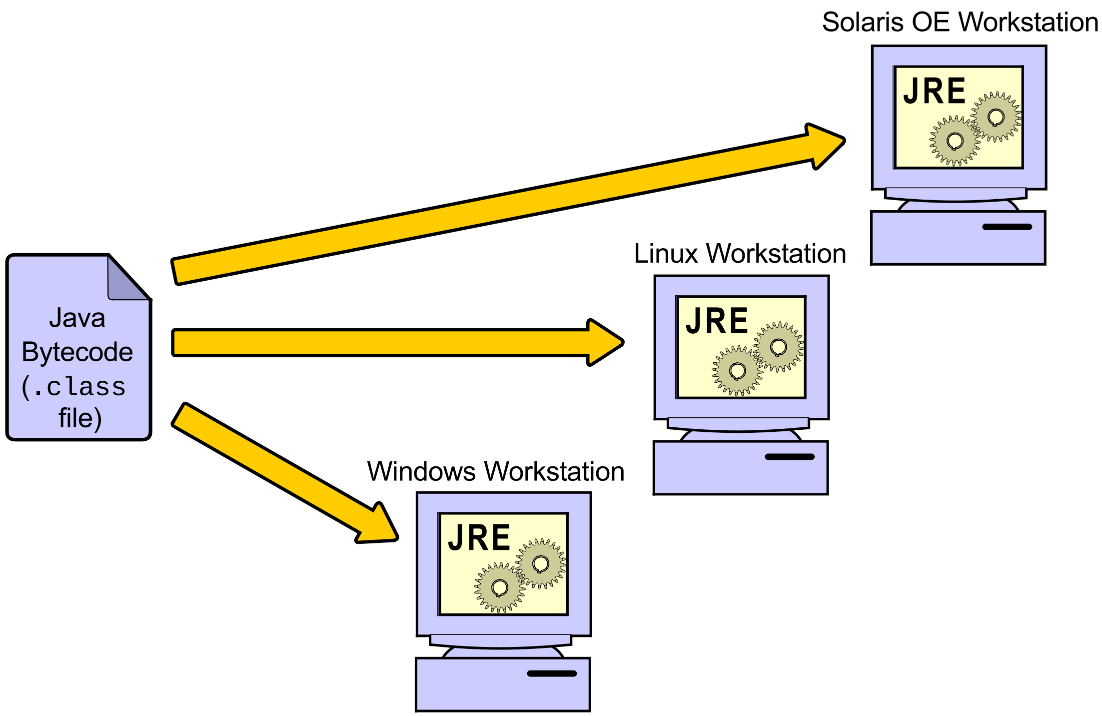
Platform-Dependent Programs



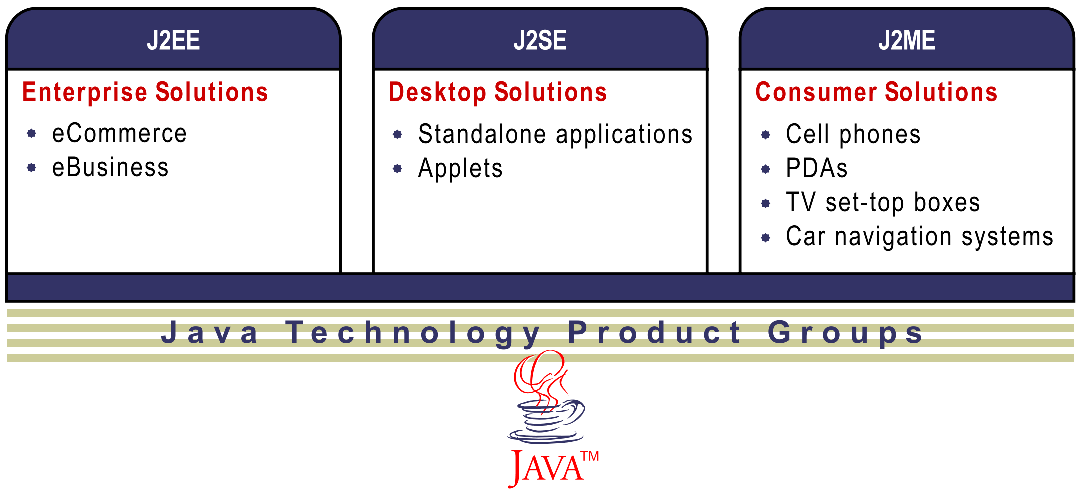
Platform-Independent Programs



Platform-Independent



Identifying Java Technology Product Groups



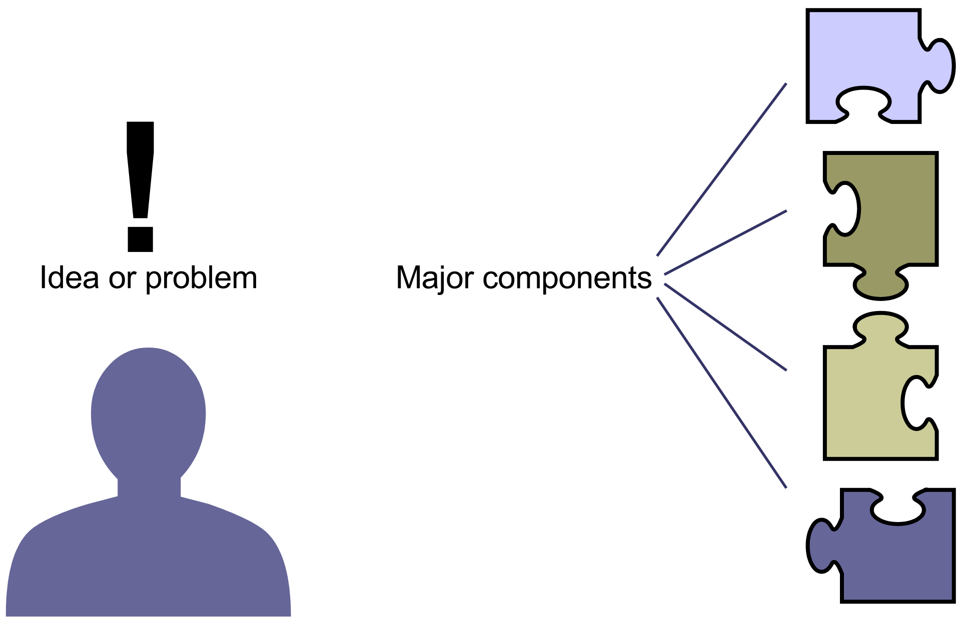
Using the Java™ 2 Platform, Standard Edition SDK Components

- Java runtime environment:
  - A Java virtual machine 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

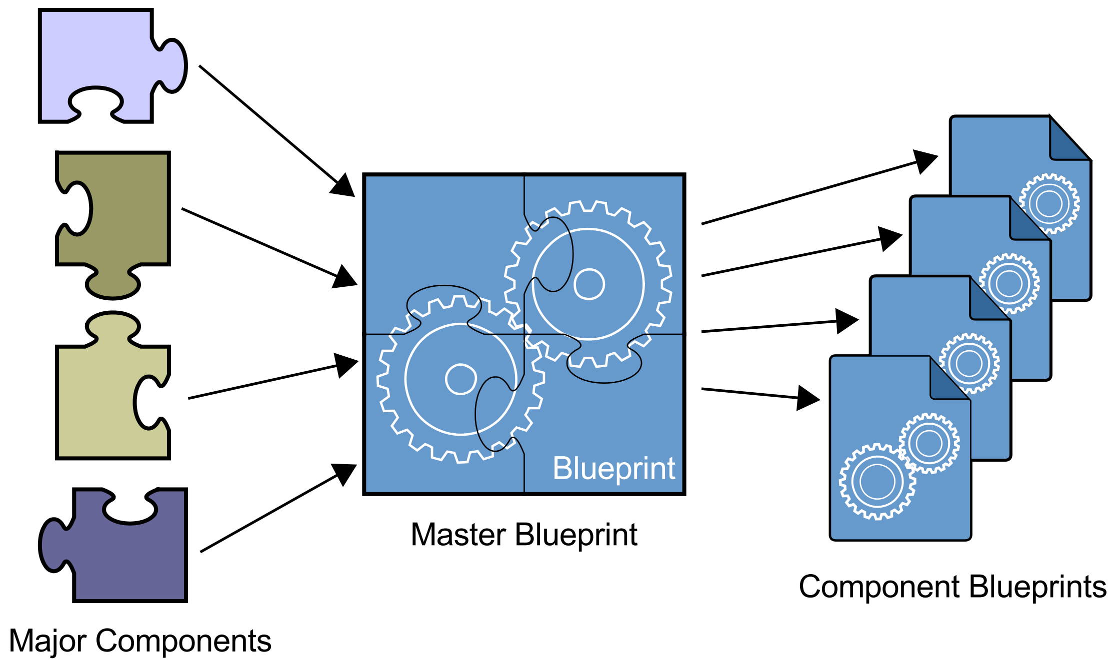
The Product Life Cycle (PLC) Stages

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

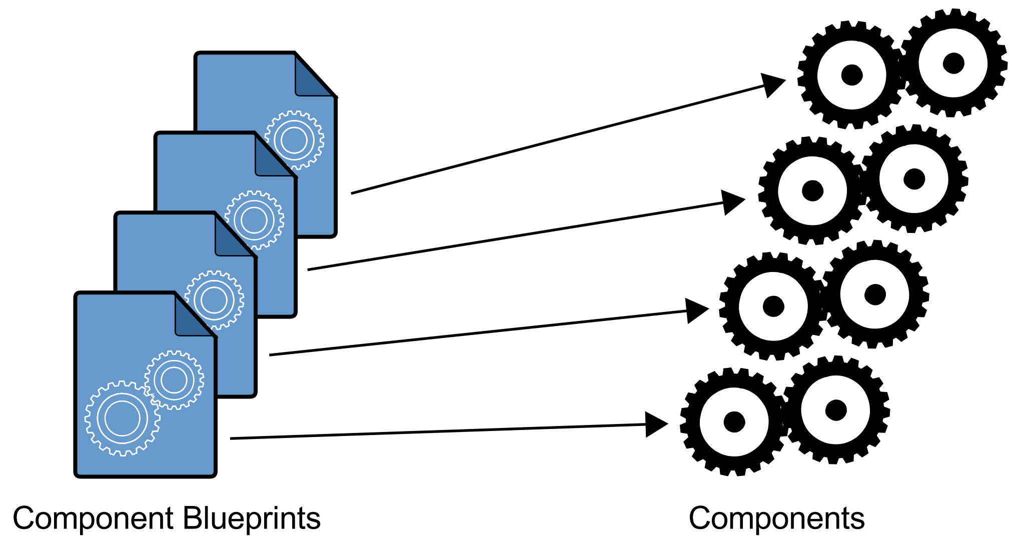
The Analysis Stage



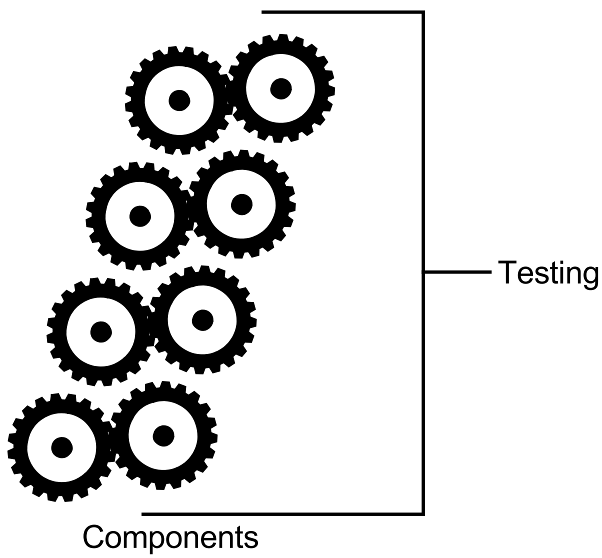
The Design Stage



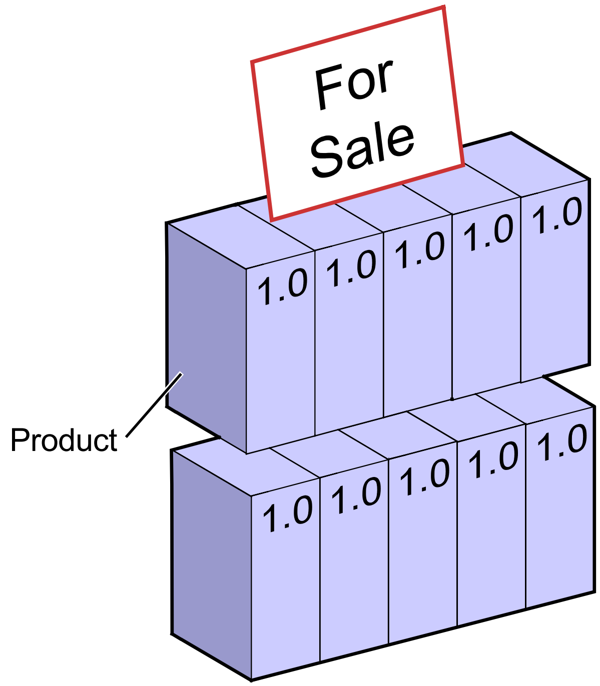
The Development Stage



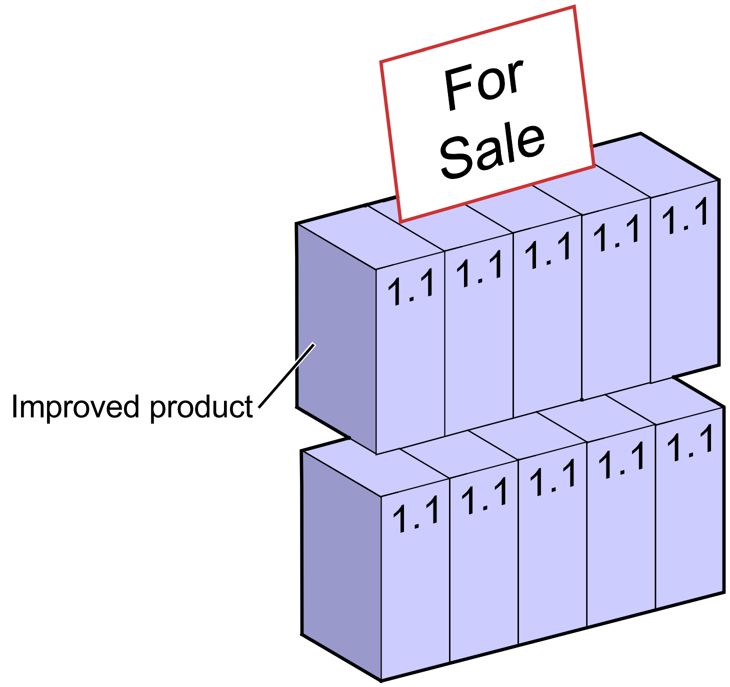
The Testing Stage



The Implementation Stage



The Maintenance Stage



The EOL Stage

Module 2

Analyzing a Problem and Designing a Solution

Overview

- Objectives:
  - Analyze a problem using object-oriented analysis
  - Design classes from which objects will be created
- Relevance

Analyzing a Problem Using Object-Oriented Analysis

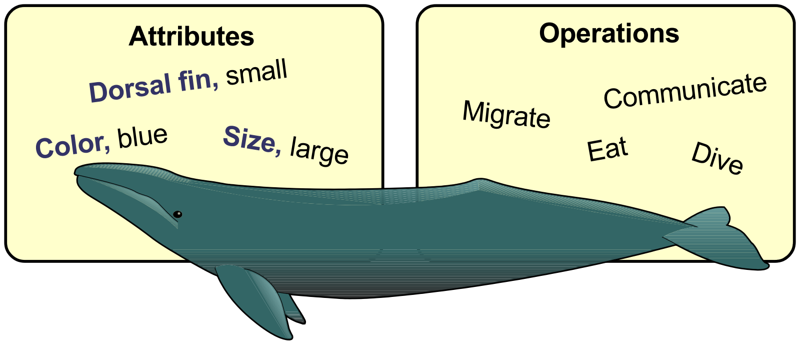
Identifying a Problem Domain

- 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.

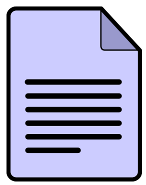
Identifying Objects



Additional Criteria for Recognizing Objects

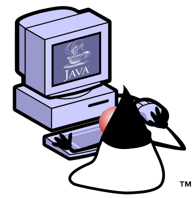
- Relevance to the problem domain:
  - Does the object exist within the boundaries of the problem domain?
  - Is the object required for the system to fulfill its responsibility?
  - Is the object required as part of an interaction between a user and the system?
- Independent existence

Possible Objects in the DirectClothing Case Study



Order

Shirt

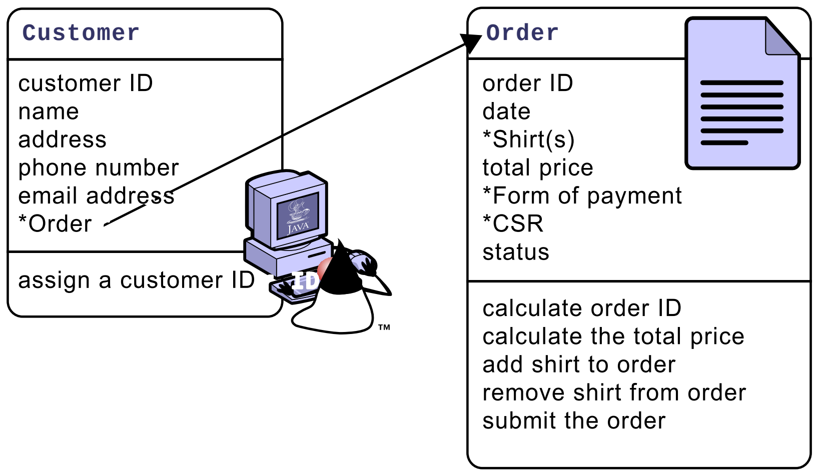


Customer

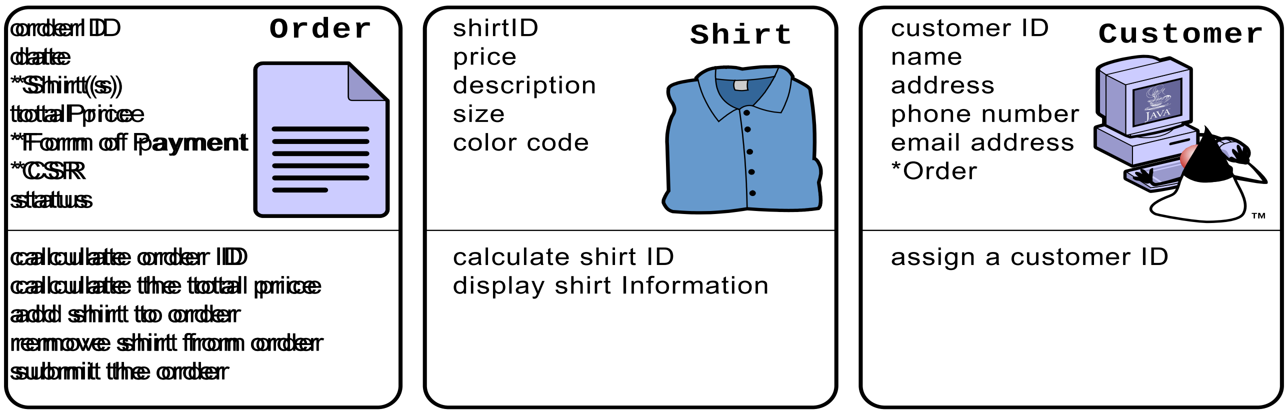
Identifying Object Attributes and Operations

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

Object With Another Object as an Attribute



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



Case Study Solution

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

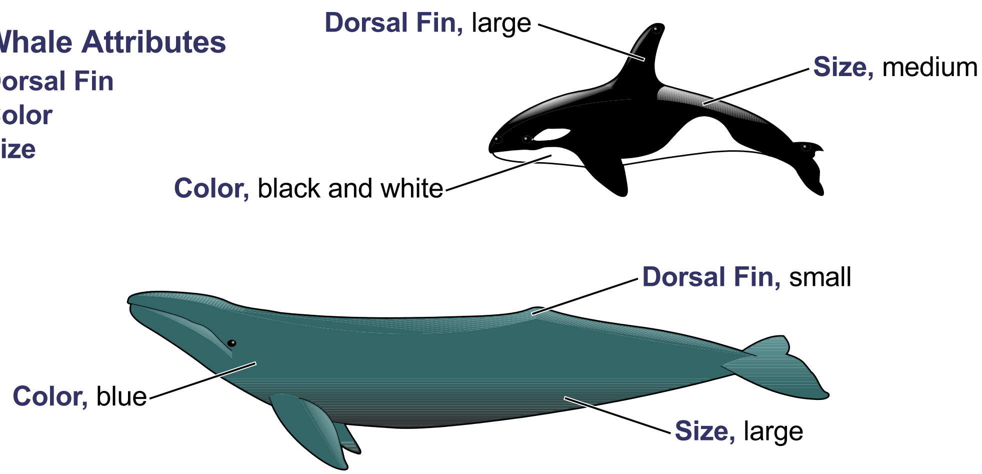
Catalog	CSR
*Shirt(s)	name extension
add a shirt remove a shirt	

Exercise 1: Analyzing a Problem Domain

- Objectives
- Tasks
- Solutions
- Discussion

Designing Classes

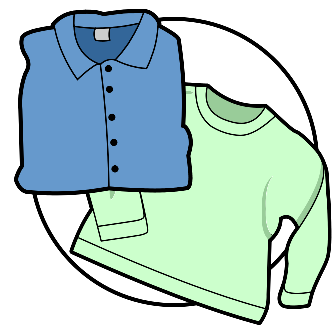
Whale Attributes
Dorsal Fin
Color
Size



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() displayShirtInformation()

Exercise 2: Designing a Solution

- Objectives
- Tasks
- Solutions
- Discussion

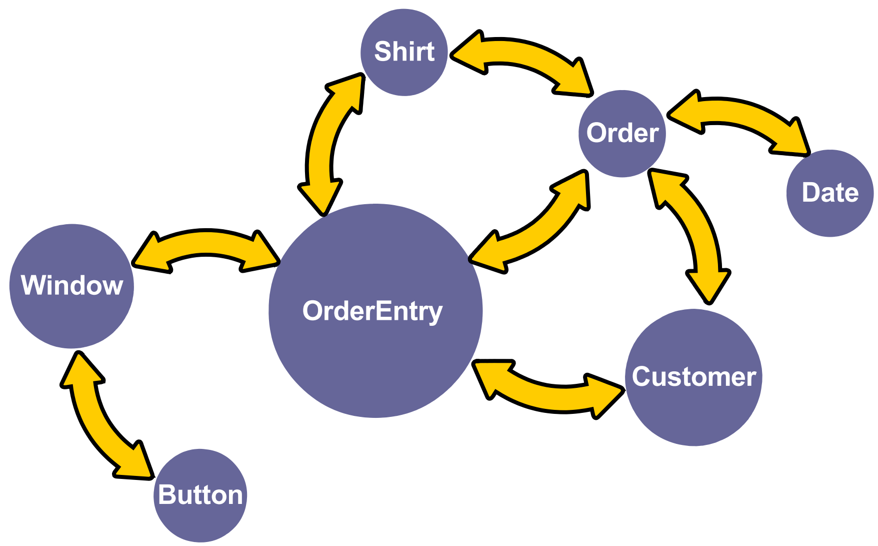
Module 3

Developing and Testing a Java Technology Program

Overview

- 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

Identifying the Components of a Class



Identifying the Components of a Class

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

Identifying the Components of a Class

```
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 displayShirtInformation() {

    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
```

The Class Declaration

- Syntax:

```
[modifier] 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 displayShirtInformation() {

    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:

```
public class ShirtTest {

  public static void main (String args[]) {

  Shirt myShirt;
  myShirt = new Shirt();

  myShirt.displayShirtInformation();

  }
}
```

The `main` Method

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 classname.java
```

   - 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 filename
```

- 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 indenting in your program, as shown in examples in this course.

Exercise: Writing, Compiling, and Testing a Basic Program

- Objectives
- Tasks
- Discussion
- Solutions

Module 4

Declaring, Initializing, and Using Variables

Overview

- 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

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 displayShirtInformation() {

    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
```

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)	2 147,334,778
`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 -2,036,854,775,808L 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 -327,456,99.01F 4.2E6F (engineering notation for $4.2 * 10^6$)
`double`	64 bits	-1111 2.1E12 999,701,327,456,99.999

```
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), such as a ‘y’.
- Example:

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

Logical Primitive Type

- The only data type is `boolean`.
- Can store only `true` or `false`.
- Used to hold 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, dashes, or any of the Java technology keywords.

Naming a Variable

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

Assigning a Value to a Variable

- Example:

```
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):

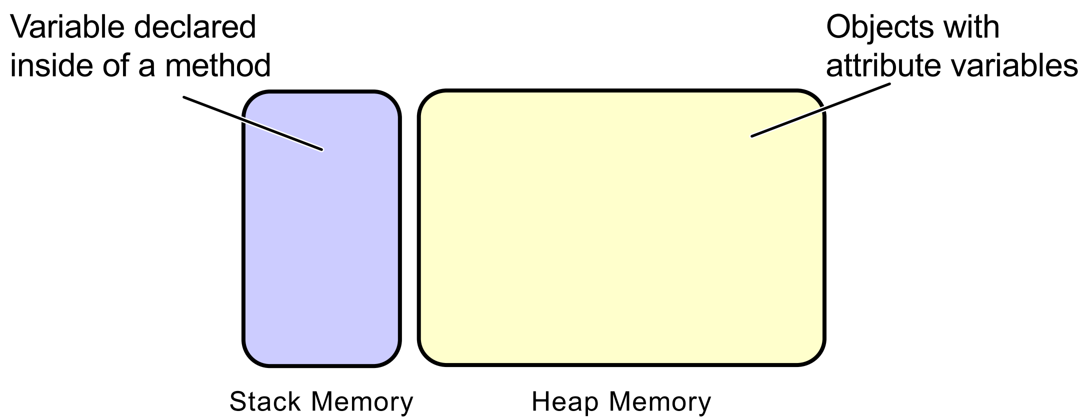
```
double salesTax = 6.25;
```

- 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



Exercise 1: Using Primitive Type Variables in a Program

- Objectives
- Tasks
- Discussion

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.	
Multiplicati on	`*`	`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).

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 R 1 6 ⟌ 31 30 1 Remainder always gives an answer with the same sign as the first operand.

Increment and Decrement Operators

- The long way:

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

- 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`.

`--`	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:

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

Is the answer 34 or 9?

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;
int small = 99L;
```

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 ;
```

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

```
int c;
```

  - Typecast 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;
```

- 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` value:

```
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
```

Exercise 2: Using Operators and Type Casting

- Objectives
- Tasks
- Discussion
- Solutions

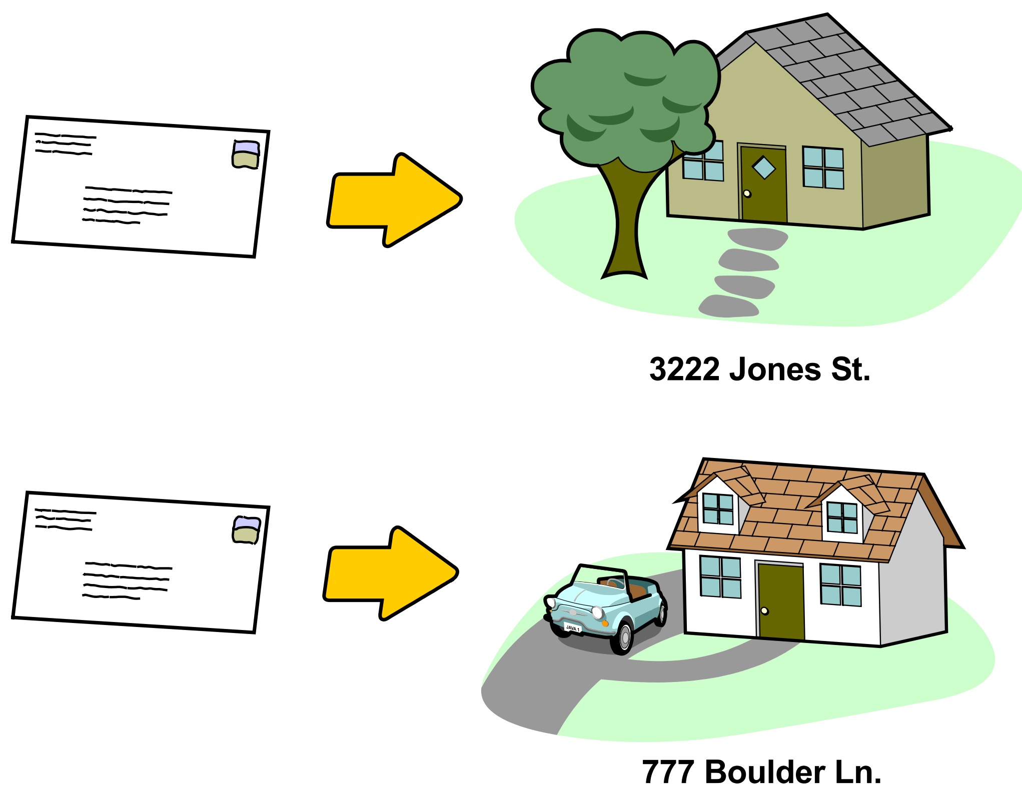
Module 5

Creating and Using Objects

Overview

- 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 2 Platform, Standard Edition™ (J2SE™) API documentation to learn about other classes in this API
- Relevance

Declaring Object References, Instantiating Objects, and Initializing Object References



Declaring Object References, Instantiating Objects, and Initializing Object References

- Example:

```
public class ShirtTest {

  public static void main (String args[]) {

  Shirt myShirt;
  myShirt = new Shirt();

  myShirt.displayShirtInformation();

  }
}
```

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();
```

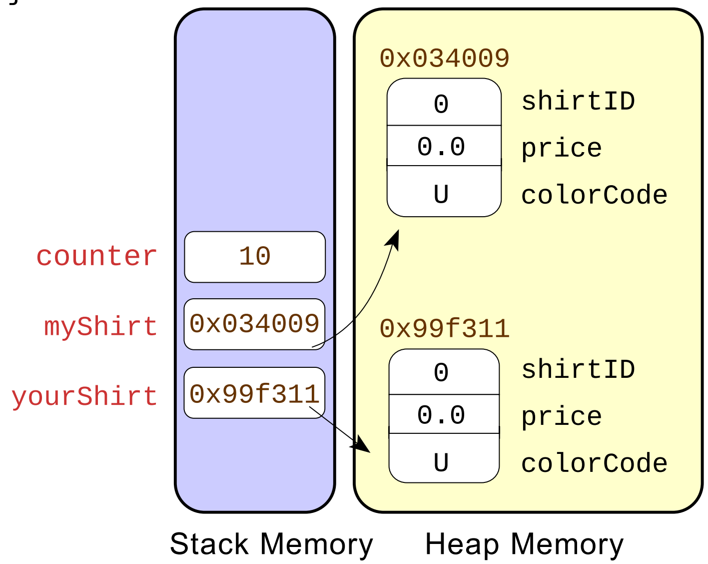
Using an Object Reference Variable to Manipulate Data

- Example:

```
public class ShirtTestTwo {

  public static void main (String args[]) {

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

  myShirt.displayShirtInformation();
  yourShirt.displayShirtInformation();

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

  myShirt.displayShirtInformation();
  yourShirt.displayShirtInformation();

  }
}

```

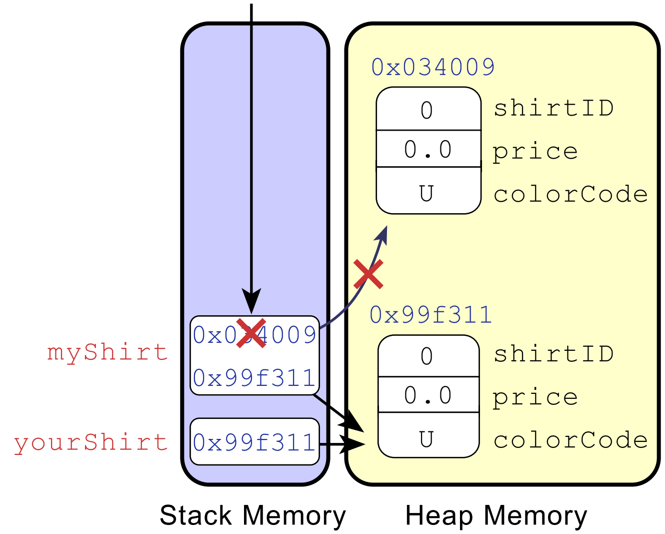
Storing Object Reference Variables in Memory

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

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



Assigning an Object Reference From One Variable to Another

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



Exercise 1: Using the ObjectTool to Create and Manipulate Objects

- Objectives
- Tasks
- Discussion

Exercise 2: Creating a Test Class

- Objectives
- Tasks
- Discussion

Using the `String` Class

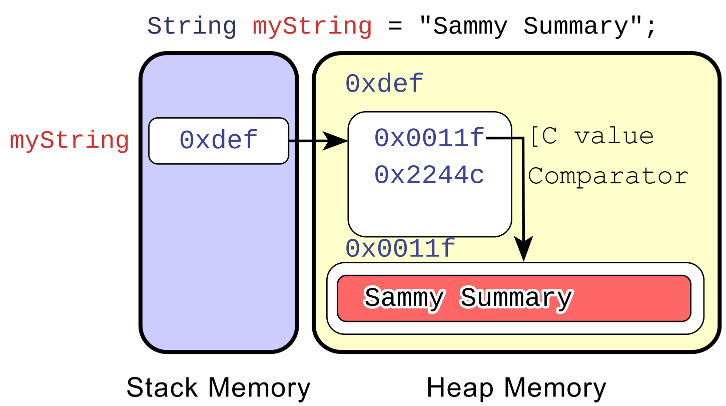
- Creating a `String` object with the `new` keyword:

```
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
```

Exercise 3: Using the `String` Class

- Objectives
- Tasks
- Discussion

Exercise 4: Examining `String` Objects With the ObjectTool

- Objectives
- Tasks
- Discussion

Investigating the Java Class Libraries

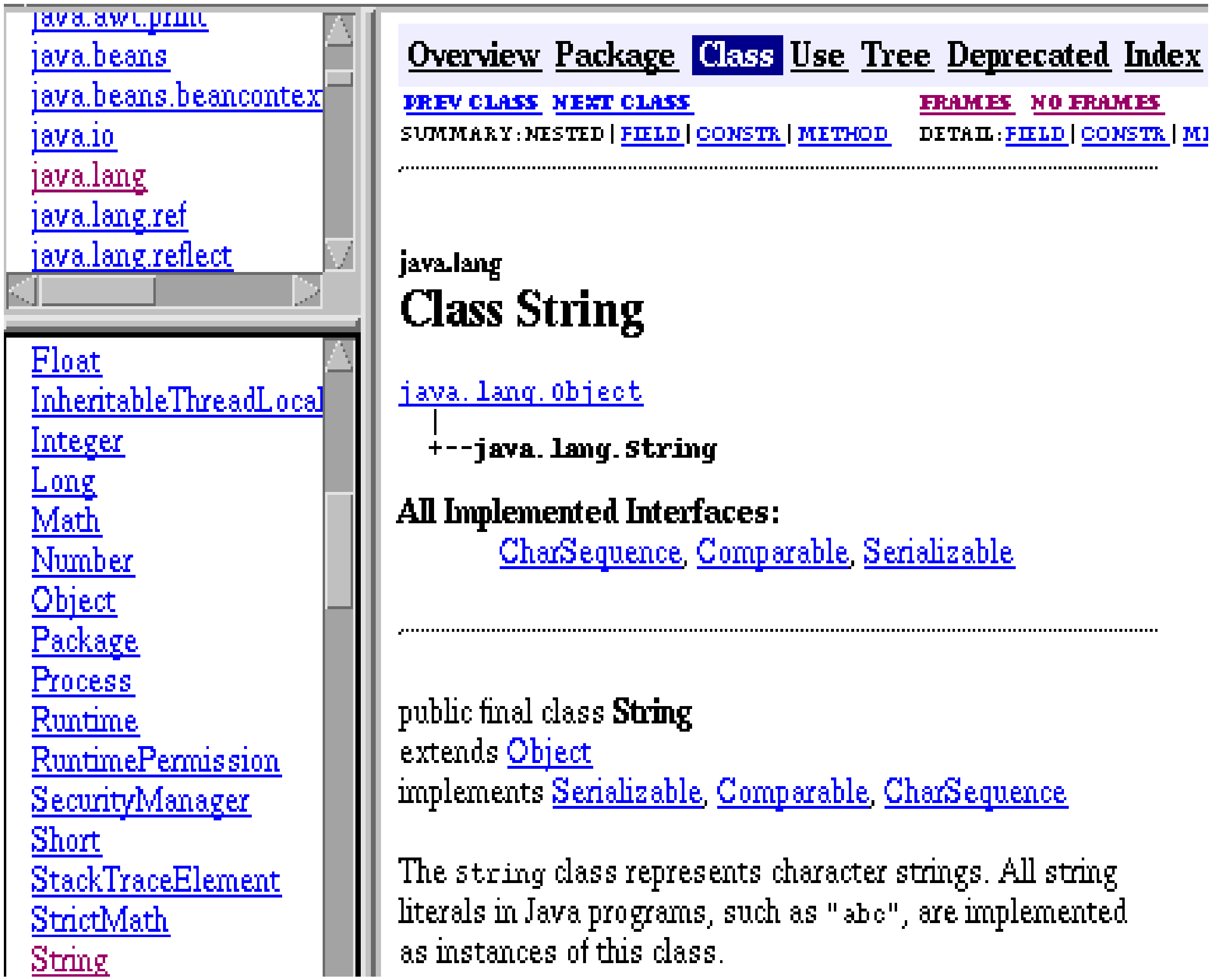
- URL to view the J2SE specification:

`http://java.sun.com/j2se/`*`version`*`/docs/api/index.html`

- Example:

`http://java.sun.com/j2se/1.4/docs/api/index.html`

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
```

Exercise 5: Using the Class Library Specification

- Objectives
- Tasks
- Discussion

Module 6

Using Operators and Decision Constructs

Overview

- Objectives:
  - Identify relational and conditional operators
  - Examine `if` and `if/else` constructs
  - Use the `switch` constructs
- Relevance

Using Relational and Conditional Operators

The 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 MAX_FLOORS = 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.");
```

```
  }

  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 int getFloor() {
    return currentFloor;
  }

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

The Elevator Test Class

```
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.closeDoor();
     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.”);
    }
  }
}
```

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;`
		`int j = 8;`
		`(!(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 MAX_FLOORS = 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.”);
```

```
    }

    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);
      }
    }

    public void setFloor() {
```

```
    int desiredFloor = 5;

    if (currentFloor < desiredFloor) {
      goUp();
    }

    if (currentFloor > desiredFloor) {
      goDown();

    }
  }

  public int getFloor() {
    return currentFloor;
  }

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

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 MAX_FLOORS = 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.”);
```

```
    }

    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) {

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

```
    }
  }

  public void setFloor() {

    int desiredFloor = 5;

    if (currentFloor < desiredFloor) {
      goUp();
    }

    if (currentFloor > desiredFloor) {
      goDown();
    }
  }

  public int getFloor() {
    return currentFloor;
  }

  public boolean checkDoorStatus() {
    return doorOpen;
```

```
    }
}
```

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 MAX_FLOORS = 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.”);
```

```
  }

  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);
    }
  }

  public void setFloor() {
```

```

    int desiredFloor = 5;

    if (currentFloor < desiredFloor) {
      goUp();
    }

    if (currentFloor > desiredFloor) {
      goDown();
    }
  }

  public int getFloor() {
    return currentFloor;
  }

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

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.”);
    }

    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.”);
  }
 }
}
```

Exercise 1: Using `if` and `if/else` Constructs

- Objectives
- Tasks
- Discussion

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;
    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

- Any equality test
- Tests against a *single* variable, such as `customerStatus`
- Tests against the value of an `int`, `short`, `byte`, or `char` type

Exercise 2: Using the `switch` Construct

- Objectives
- Tasks
- Discussion

Module 7

Using Loop Constructs

Overview

- Objectives:
  - Create `while` loops
  - Develop `for` loops
  - Create `do/while` loops
- Relevance

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 int weight = 0;

  public final int CAPACITY = 1000;
  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.");
      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 = 5;

      while (currentFloor != desiredFloor)
```

```
      if (currentFloor < desiredFloor) {
        goUp();
      }
      else {
        goDown();
      }
  }

  public int getFloor() {
    return currentFloor;
  }

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

Nested `while` Loops

- Problem:

```
@@@@@@@@@@
@@@@@@@@@@
@@@@@@@@@@
```

- 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++;
    }
  }
}
```

Exercise 1: Using the `while` Loop

- Objectives
- Tasks
- Discussion

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 int weight = 0;

  public final int CAPACITY = 1000;
  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.");
      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 = 5;

      if (currentFloor > desiredFloor) {
```

```
      for (int down = currentFloor; down != desiredFloor; --down) {
        goDown();
      }
    }

    else {
      for (int up = currentFloor; up != desiredFloor; ++up) {
        goUp();
      }
    }
  }

  public int getFloor() {
    return currentFloor;
  }

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

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();
    }
  }
}
```

Exercise 2: Using the `for` Loop

- Objectives
- Tasks
- Discussion

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 int weight = 0;

  public final int CAPACITY = 1000;
  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.");
    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 = 5;

    do {
```

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

  public int getFloor() {
    return currentFloor;
  }

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

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++;
      }
```

```
        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.

Exercise 3: Using the `do/while` Loop

- Objectives
- Tasks
- Discussion

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

Creating and Invoking Methods

- Syntax:

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

The Basic Form of a Method

Example:

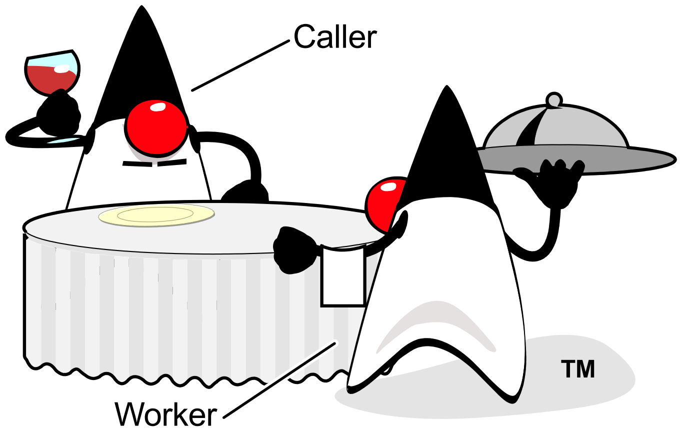
```
public void displayShirtInformation() {
   System.out.println("Shirt ID: " + shirtID);
    System.out.println("Shirt description:" + description);
   System.out.println("Color Code: " + colorCode);
    System.out.println("Shirt price: " + price);
    System.out.println("Quantity in stock: " + quantityInStock);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

  }
}

```

Calling and Worker Methods



Invoking a Method in the Same Class

- Example

```
public class Elevator {

  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;

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

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

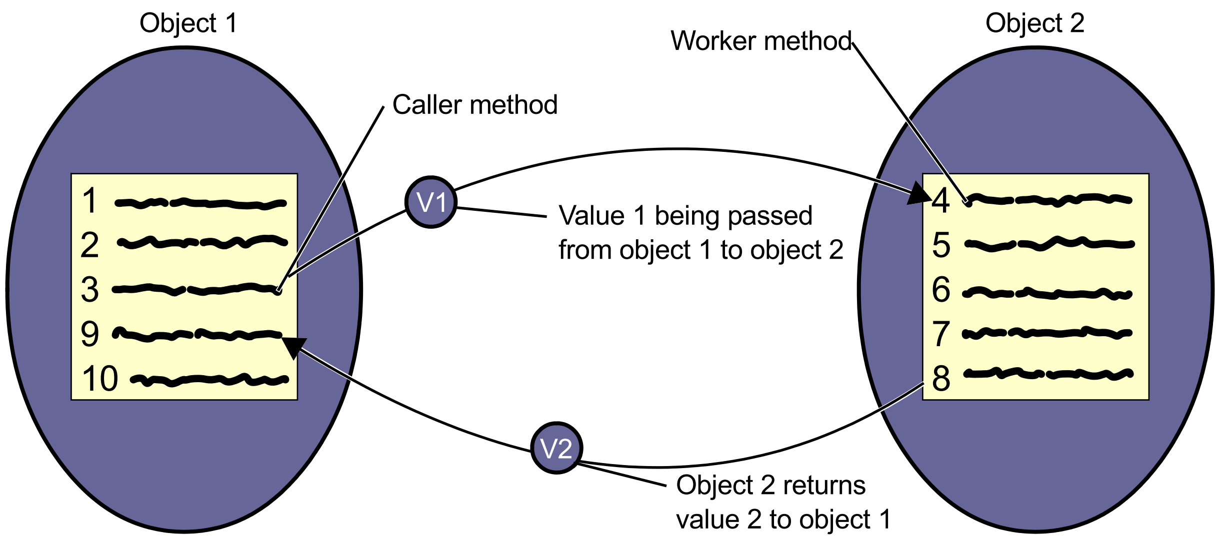
```
    doorOpen = false;
    System.out.println(“Door is closed.”);
  }

  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) {
        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();
     myElevator.goDown();
```

```

     int curFloor = myElevator.getFloor();

     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 getSum() {
    return sum;
  }
```

Receiving Return Values

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();
     myElevator.goDown();
```

```

      int curFloor = myElevator.getFloor();

      myElevator.setFloor(myElevator.TOP_FLOOR);
      }
      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.

Exercise 1: Using Arguments and Return Values

- Objectives
- Tasks
- Discussion

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

- Declaring `static` variables:

```
static double SALES_TAX = 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
  - The `StrictMath` 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 associate 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;
  }

  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()`	Terminate the current line by writing the line separator string
`void println(boolean x)`	Print a `boolean value` and then terminate the line
`void println(char x)`	Print a character and then terminate the line
`void println(char[] x)`	Print an array of characters and then terminate the line
`void println(double x)`	Print a `double` and then terminate the line
`void println(float x)`	Print a `float` and then terminate the line
`void println(int x)`	Print an `int` and then terminate the line
`void println(long x)`	Print a `long` and then terminate the line
`void println(Object x)`	Print an object and then terminate the line
`void println(String x)`	Print a string and then terminate 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 d, double p){
    shirtID = ID;
    description = d;
    price = p;
  }

  public void setShirtInfo(int ID, String d, double p, char c){
```

```
    shirtID = ID;
    description = d;
    colorCode = c;
    price = p;
  }

  public void setShirtInfo(int ID, String d, double p, char c, int q){
    shirtID = ID;
    description = d;
    colorCode = c;
    price = p;
    quantityInStock = q;
  }

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

    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();
  }
}
```

Exercise 2: Using Overloaded Methods

- Objectives
- Tasks
- Discussion

Module 9

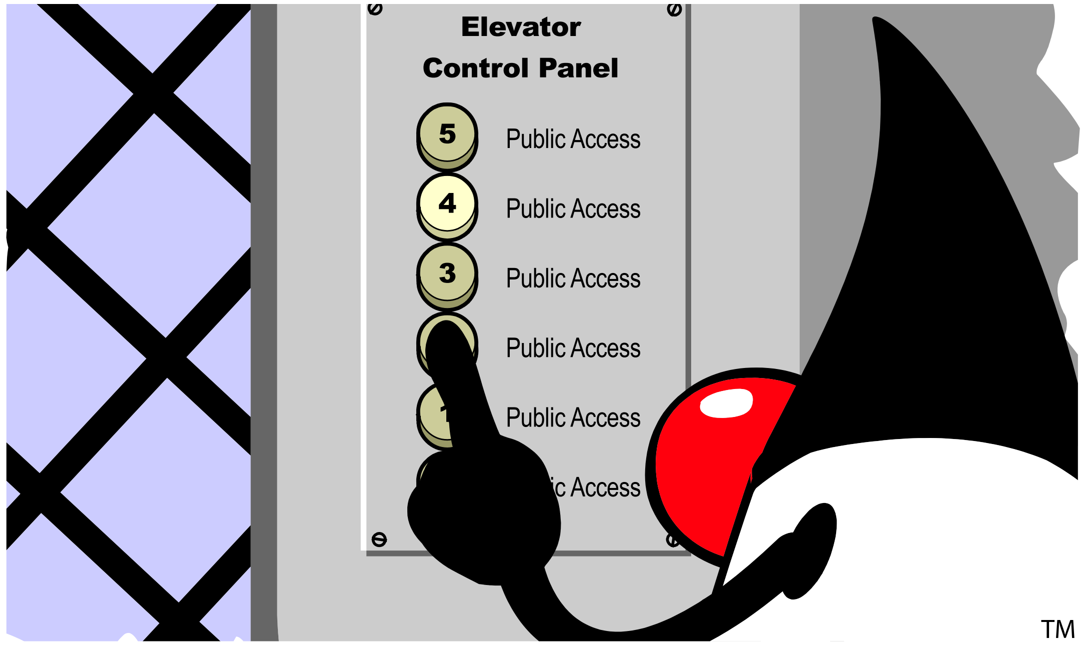
Implementing Encapsulation and Constructors

Overview

- Objectives:
  - Use encapsulation to protect data
  - Create constructors to initialize objects
- Relevance

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

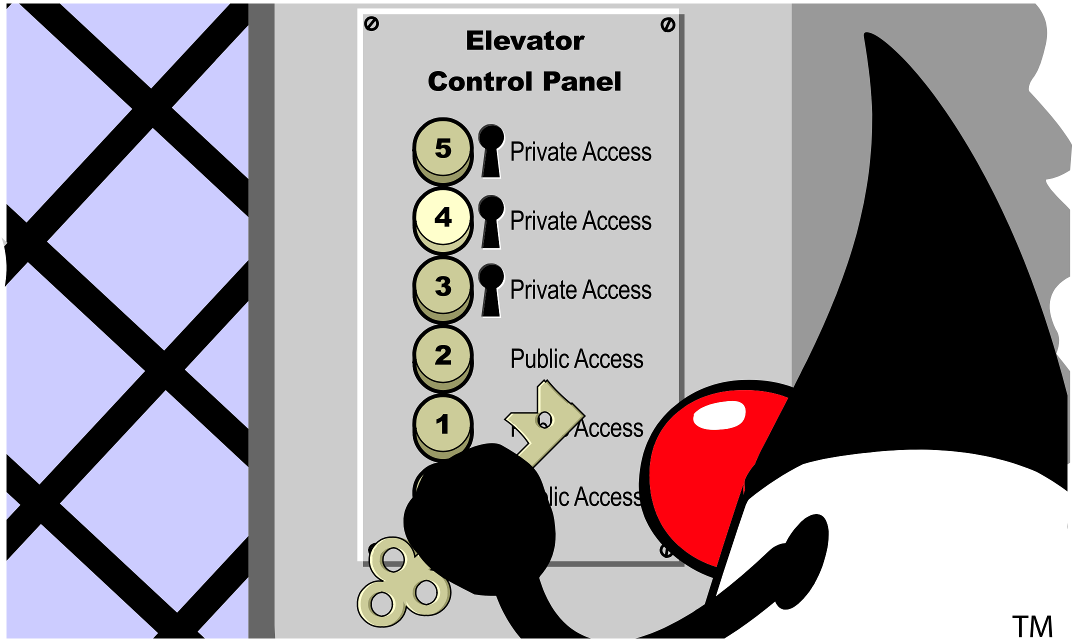
The public Modifier

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;
    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 final int TOP_FLOOR = 5;
   private final int BOTTOM_FLOOR = 1;
}
```

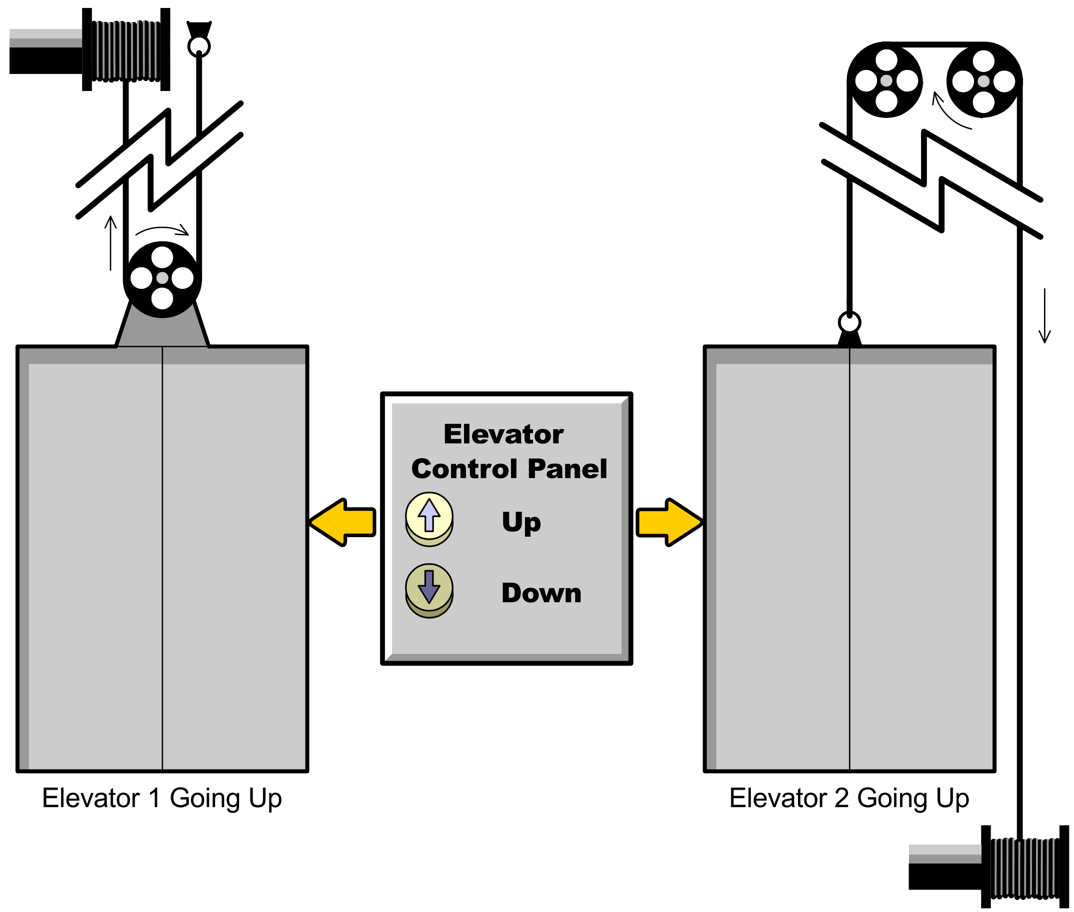
The `private` Modifier

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--;
    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;
  }
```

```

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

} // end of class
```

Interface and Implementation

Example:

```
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');
    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) {

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

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

```

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

An Encapsulated Elevator

Example:

```
public class PrivateElevator2 {

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

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

  public void openDoor() {
    doorOpen = true;
  }

  public void closeDoor() {
    calculateCapacity();

```

```
    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);
  }

  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) {
    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();
}

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

```

  public int getFloor() {
    return currentFloor;
  }

  public boolean getDoorStatus() {
    return doorOpen;
  }
}

```

An 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();
     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
```

Exercise 1: Writing Encapsulated Classes

- Objectives
- Tasks
- Discussion

Describing Variable Scope

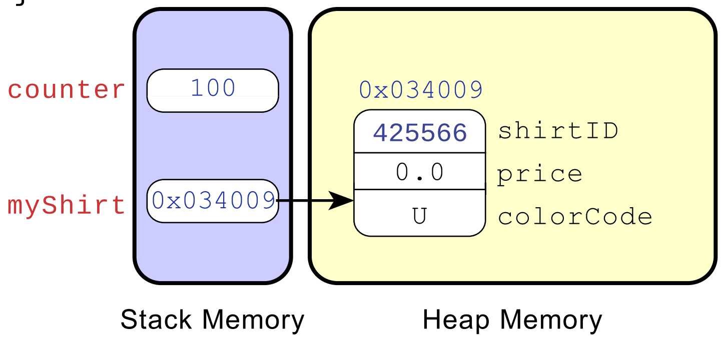
Example:

```
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 {

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

- 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’:
      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);
  }
}
```

The Default Constructor

- Example:

```
ConstructorShirt constShirt = new ConstructorShirt();
```

- 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’;
  }

```

```
  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) {

    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;
      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;
    }


  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();
    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);
    }
}
```

Exercise 2: Using Constructors

- Objectives
- Tasks
- Discussion

Module 10

Creating and Using Arrays

Overview

- 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

Creating One-Dimensional Arrays

Example:

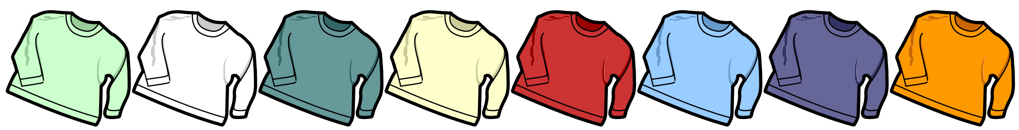
```
int ageOne = 27;
int ageTwo = 12;
int ageThree = 82;
int ageFour = 70;
int ageFive = 54;
int ageSix = 6;
int ageSeven = 1;
int ageEight = 30;
int ageNine = 34;
int ageTen = 42;
```

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(121,"Work Shirt", 'B', 12.95),
   new Shirt(122,"Flannel Shirt", 'G', 22.95)};

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

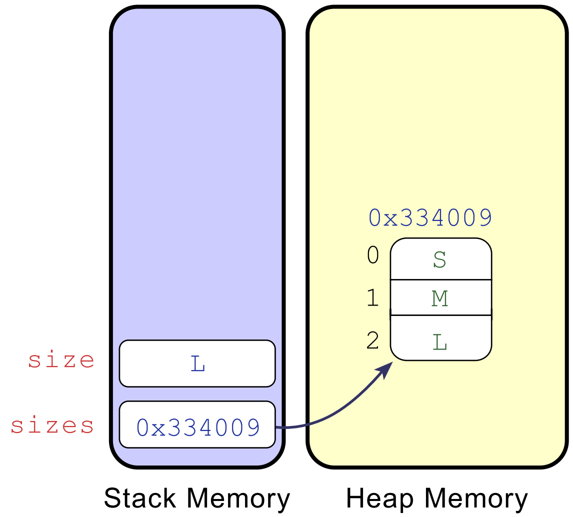
Accessing a Value Within an Array

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];
```

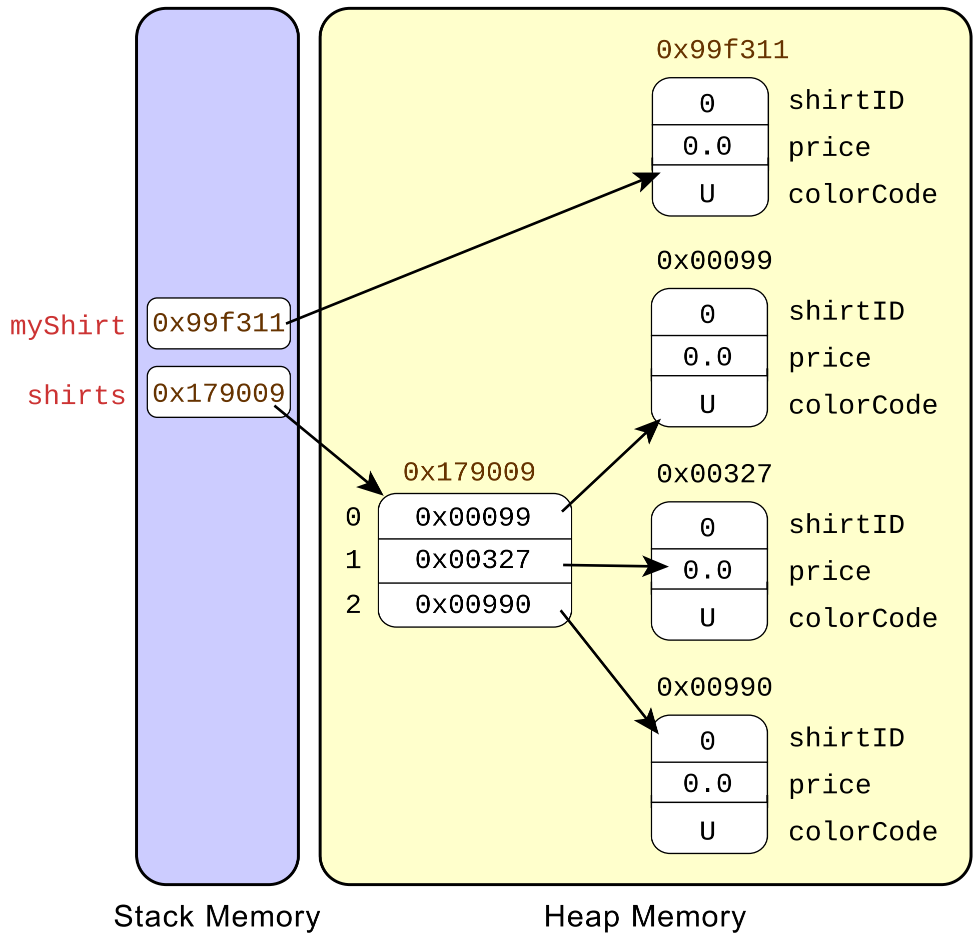
Storing Primitive Variables and Arrays of Primitives in Memory

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



Storing Reference Variables and Arrays of References in Memory

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



Exercise 1: Creating and Using One-Dimensional Arrays

- Objectives
- Tasks
- Discussion

Exercise 2: Viewing Arrays Using the ObjectTool

- Objectives
- Tasks
- Discussion

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;
}
```

Exercise 3: Using Loops and Arrays

- Objectives
- Tasks
- Discussion

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

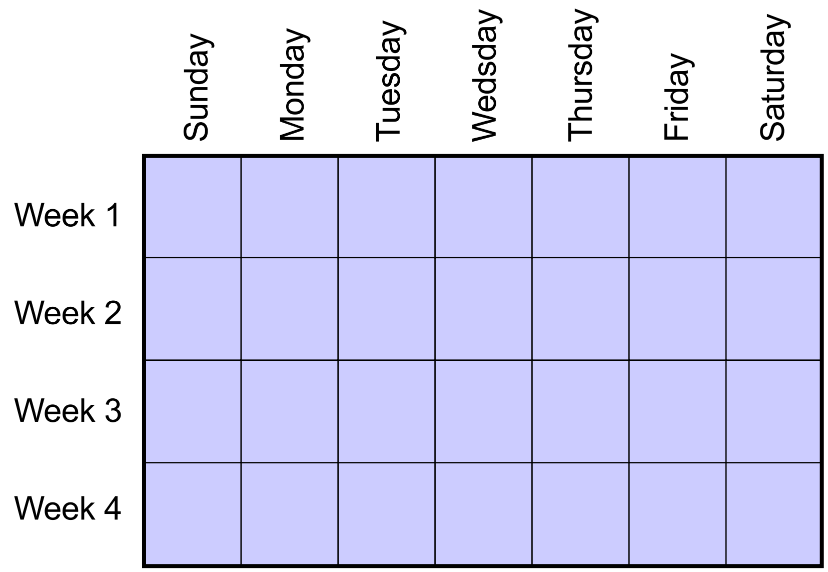
Example:

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

Exercise 4: Parsing the `args[]` Array

- Objectives
- Tasks
- Discussion

Describing Two-Dimensional Arrays



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				

Exercise 5: Creating and Using Two-Dimensional Arrays

- Objectives
- Tasks
- Discussion

Module 11

Implementing Inheritance

Overview

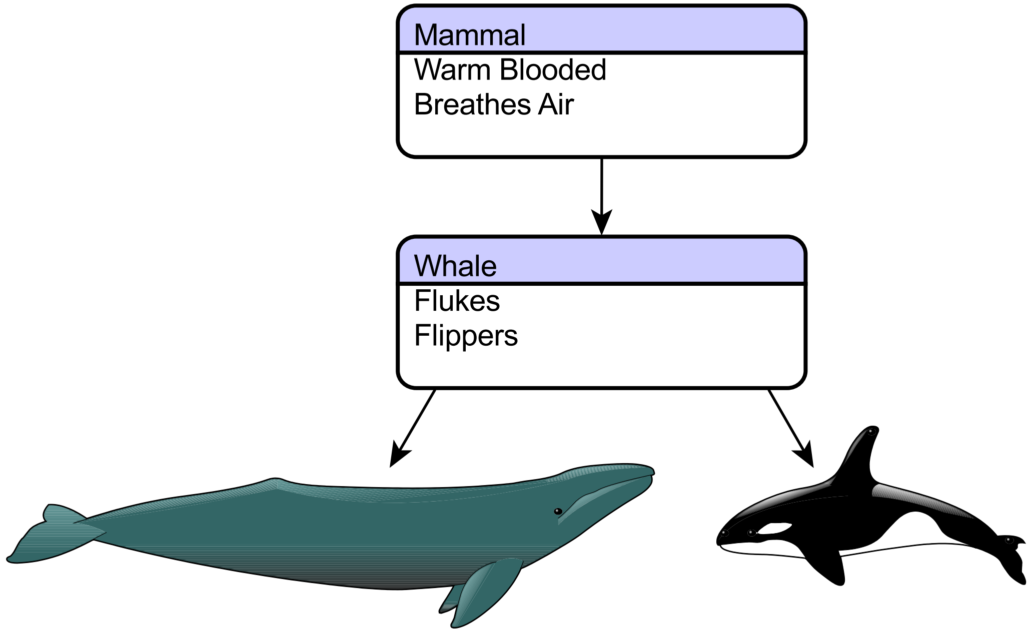
- Objectives:
  - Define and test your use of inheritance
  - Explain abstraction
  - Explicitly identify class libraries used in your code
- Relevance

Inheritance

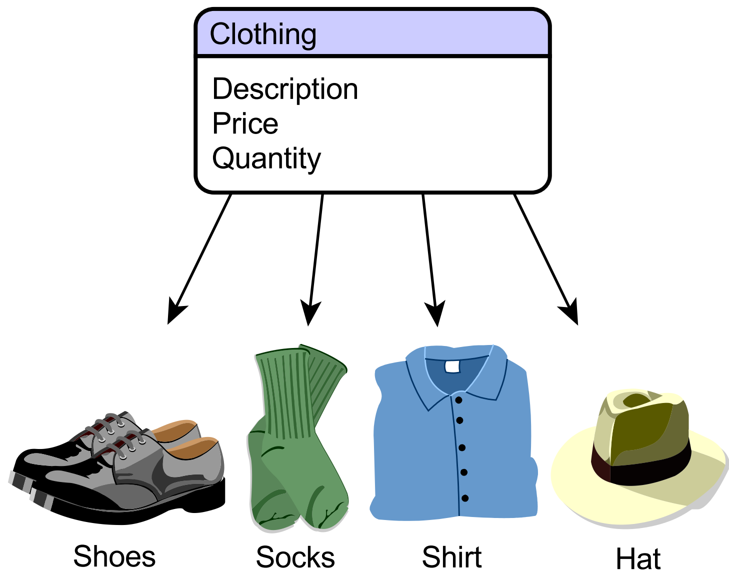
Hat	Sock
hatID price description colorCode R=Red, B=Blue, G=Green quantityInStock	sockID price description colorCode R=Red, B=Blue, G=Green quantityInStock
calculateHatID() displayHatInformation()	calculateSockID() displaySocksInformation()

Pant	Shirt
pantID price size gender M=Male, F=Female description colorCode B=Blue, T=Tan quantityInStock	shirtID price description colorCode R=Red, B=Blue, G=Green quantityInStock
calculatePantID() displayPantInformation()	calculateShirtID() displayShirtInformation()

Superclasses and Subclasses



Testing Superclass and Subclass Relationships



Modeling Superclasses and Subclasses

Hat:Clothing	Sock:Clothing
colorCode R=Red, B=Blue, G=Green	colorCode R=Red, B=Blue, G=Green
displayHatInformation()	displaySockInformation()

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

Modeling Superclasses and Subclasses

Clothing
ID price description quantityInStock
calculateID()

Declaring a Subclass

- Syntax:

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

- Example:

```
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 displayShirtInformation() {

    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
```

Declaring a Subclass

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

  // This method displays the values for an item
  public void calculateID() {
    int uniqueID = 0;
    ++uniqueID;

    ID = uniqueID;
  }

  public int getID() {
```

```
    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;
  }

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

```

  public int getQuantityInStock() {
    return quantityInStock;
  }

} // end of class
```

Exercise 1: Creating Superclasses and Subclasses

- Objectives
- Tasks
- Discussion

Exercise 2: Viewing Class Hierarchies Using the ObjectTool

- Objectives
- Tasks
- Discussion

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
}
```

Course Contents

About This Course .... Preface-iv

Course Goals .... Preface-v
Course Map .... Preface-vi
Topics Not Covered .... Preface-vii
How Prepared Are You? .... Preface-viii
Introductions .... Preface-ix
How to Use the Icons .... Preface-x
Typographical Conventions .... Preface-xii
Additional Conventions .... Preface-xiv

Explaining Java™ Technology 1-1

Overview 1-2
Key Concepts of the Java Programming Language 1-3
Object-Oriented 1-4
Distributed 1-6
Simple 1-7
Multithreaded 1-8
Secure 1-9
Platform-Dependent Programs 1-10
Platform-Independent 1-14
Identifying Java Technology Product Groups 1-15
Using the Java™ 2 Platform, Standard Edition SDK Components 1-16
The Product Life Cycle (PLC) Stages 1-17
The Analysis Stage 1-18
The Design Stage 1-19
The Development Stage 1-20
The Testing Stage 1-21

The Implementation Stage 1-22
The Maintenance Stage 1-23
The EOL Stage 1-24

Analyzing a Problem and Designing a Solution 2-1
Overview 2-2
Analyzing a Problem Using Object-Oriented Analysis 2-3
Identifying a Problem Domain 2-4
Identifying Objects 2-5
Additional Criteria for Recognizing Objects 2-7
Possible Objects in the DirectClothing Case Study 2-8
Identifying Object Attributes and Operations 2-9
Object With Another Object as an Attribute 2-10
Possible Attributes and Operations in the DirectClothing, Inc. Case Study 2-11
Case Study Solution 2-12
Exercise 1: Analyzing a Problem Domain 2-15
Designing Classes 2-16
Class and Resulting Objects 2-17
Modeling Classes 2-18
Exercise 2: Designing a Solution 2-19

Developing and Testing a Java Technology Program 3-1
Overview 3-2
Identifying the Components of a Class 3-3
Identifying the Components of a Class 3-4
The Class Declaration 3-7
Variable Declarations and Assignments 3-8
Comments 3-9
Methods 3-10

Creating and Using a Test Class 3-11
The `main` Method 3-12
Compiling a Program 3-13
Executing (Testing) a Program 3-14
Debugging Tips 3-15
Exercise: Writing, Compiling, and Testing a Basic Program 3-16

Declaring, Initializing, and Using Variables 4-1
Overview 4-2
Identifying Variable Use and Syntax 4-3
Uses for Variables 4-5
Variable Declaration and Initialization 4-6
Describing Primitive Data Types 4-7
Integral Primitive Types 4-8
Floating Point Primitive Types 4-10
Textual Primitive Type 4-11
Logical Primitive Type 4-12
Naming a Variable 4-13
Assigning a Value to a Variable 4-15
Declaring and Initializing Several Variables in One Line of Code 4-16
Additional Ways to Declare Variables and Assign Values to Variables 4-17
Constants 4-19
Storing Primitives and Constants in Memory 4-20
Exercise 1: Using Primitive Type Variables in a Program 4-21
Standard Mathematical Operators 4-22
Increment and Decrement Operators 4-24
Operator Precedence 4-27
Using Parentheses 4-28
Using Promotion and Type Casting 4-29

Promotion 4-30
Type Casting 4-31
Compiler Assumptions for Integral and Floating Point Data Types 4-33
Floating Point Data Types and Assignment 4-34
Example 4-35
Exercise 2: Using Operators and
Type Casting 4-36

Creating and Using Objects 5-1
Overview 5-2
Declaring Object References, Instantiating Objects, and Initializing Object References 5-3
Declaring Object Reference Variables 5-5
Instantiating an Object 5-6
Initializing Object Reference Variables 5-7
Using an Object Reference Variable to Manipulate Data 5-8
Storing Object Reference Variables in Memory 5-9
Assigning an Object Reference From One Variable to Another 5-10
Exercise 1: Using the ObjectTool to Create and Manipulate Objects 5-11
Exercise 2: Creating a Test Class 5-12
Using the `String` Class 5-13
Storing `String` Objects in Memory 5-14
Using Reference Variables for
`String` Objects 5-15
Exercise 3: Using the `String` Class 5-16
Exercise 4: Examining `String` Objects With the ObjectTool 5-17
Investigating the Java Class Libraries 5-18
Using the Java Class Library Specification to Learn About a Method 5-20
Exercise 5: Using the Class Library Specification 5-21

Using Operators and Decision Constructs 6-1
Overview 6-2
Using Relational and Conditional Operators 6-3
The Elevator Example 6-4
The Elevator Test Class 6-6
Relational Operators 6-7
Testing Equality Between Strings 6-8
Conditional Operators 6-9
The `if` Construct 6-10
Nested `if` Statements 6-15
The `if/else` Construct 6-19
The `if/else` Construct 6-20
Chaining `if/else` Constructs 6-24
Exercise 1: Using `if` and `if/else` Constructs 6-27
Using the `switch` Construct 6-28
When to Use `switch` Constructs 6-31
Exercise 2: Using the `switch` Construct 6-32

Using Loop Constructs 7-1
Overview 7-2
Creating `while` Loops 7-3
Nested `while` Loops 7-7
Exercise 1: Using the `while` Loop 7-9
Developing a `for` Loop 7-10
Nested `for` Loops 7-14
Exercise 2: Using the `for` Loop 7-15
Coding a `do/while` Loop 7-16
Nested `do/while` Loops 7-20
Comparing Loop Constructs 7-22
Exercise 3: Using the `do/while` Loop 7-23

Developing and Using Methods 8-1
Overview 8-2
Creating and Invoking Methods 8-3
The Basic Form of a Method 8-4
Invoking a Method From a Different Class 8-5
Calling and Worker Methods 8-6
Invoking a Method in the Same Class 8-7
Guidelines for Invoking Methods 8-10
Passing Arguments and Returning Values 8-11
Declaring Methods With Arguments 8-12
The `main` Method 8-13
Invoking Methods With Arguments 8-14
Declaring Methods With Return Values 8-16
Returning a Value 8-17
Receiving Return Values 8-18
Advantages of Method Use 8-20
Exercise 1: Using Arguments and Return Values 8-21
Creating `Static` Methods and Variables 8-22
Static Methods and Variables in the Java API 8-25
Using Method Overloading 8-27
Method Overloading and the Java API 8-30
Uses for Method Overloading 8-31
Exercise 2: Using Overloaded Methods 8-36

Implementing Encapsulation and Constructors 9-1
Overview 9-2
Using Encapsulation 9-3
The `public` Modifier 9-4
The `private` Modifier 9-8
Interface and Implementation 9-12

An Encapsulated Elevator 9-21
Sample Output 9-28
Exercise 1: Writing Encapsulated Classes 9-29
Describing Variable Scope 9-30
How Instance Variables and Local Variables Appear in Memory 9-32
Creating Constructors 9-33
The Default Constructor 9-36
Overloading Constructors 9-38
Exercise 2: Using Constructors 9-43

Creating and Using Arrays 10-1
Overview 10-2
Creating One-Dimensional Arrays 10-3
Declaring a One-Dimensional Array 10-5
Instantiating a One-Dimensional Array 10-6
Initializing a One-Dimensional Array 10-7
Declaring, Instantiating, and Initializing One-Dimensional Arrays 10-8
Accessing a Value Within an Array 10-9
Storing Primitive Variables and Arrays of Primitives in Memory 10-10
Storing Reference Variables and Arrays of References in Memory 10-11
Exercise 1: Creating and Using
One-Dimensional Arrays 10-12
Exercise 2: Viewing Arrays Using the ObjectTool 10-13
Setting Array Values Using the `length` Attribute and a Loop 10-14
Exercise 3: Using Loops and Arrays 10-15
Using the `args` Array in the `main` Method 10-16
Converting `String` Arguments to Other Types 10-17
Exercise 4: Parsing the `args[]` Array 10-18
Describing Two-Dimensional Arrays 10-19
Declaring a Two-Dimensional Array 10-20

Instantiating a Two-Dimensional Array 10-21
Initializing a Two-Dimensional Array 10-22
Exercise 5: Creating and Using
Two-Dimensional Arrays 10-23

Implementing Inheritance 11-1
Overview 11-2
Inheritance 11-3
Superclasses and Subclasses 11-4
Testing Superclass and Subclass Relationships 11-5
Modeling Superclasses and Subclasses 11-6
Declaring a Subclass 11-8
Exercise 1: Creating Superclasses and Subclasses 11-12
Exercise 2: Viewing Class Hierarchies Using the ObjectTool 11-13
Abstraction 11-14
Classes in the Java API 11-15
The `import` Statement 11-16
Specifying the Fully Qualified Name 11-17