Introduction to Java and OOA/OOD for Web Applications

Alvin J. Alexander
devdaily.com

Contents

Chapter 1

Day 1: Object-Oriented Software Development

1.1 Credits and Other Material

This training material about the Java Programming Language and Object-Oriented Programming methods is always used with at least two other books, to whom we owe a great deal of credit. These books are:

1. The Java Programming Language, by Arnold and Gosling
2. Use Case Driven Object Modeling With Uml: A Practical Approach, by Rosenberg and Scott

Many other books and reference materials have been used in the creation of this material, and they are all listed in the bibliography at the end of this material.

1.2 Why OO?

1.2.1 Benefits of object-oriented programming

Why has object-oriented programming gone from being "something to think about" to being a de-facto standard in the way software is developed today? OOA/OOD/OOP is good for:

- Analyzing user requirements
- Designing software
- Constructing software
  - Reusability (reusable components)
  - Reliability
  - Robustness
  - Extensibility
  - Maintainability
- Reducing large problems to smaller, more manageable problems

According to the GartnerInstitute ...

- 74% of all IT projects fail, come in over budget, or run past the original deadline.
- 28% fail altogether.
- 52.7% of IT projects cost 189
- Every year $75B is spent on failed IT projects.

1.2.2 Symptoms of software development problems

The text "Rational Unified Process, An Introduction" [9] identifies the following symptoms that characterize failing software development projects.

- Inaccurate understanding of end-user needs.
- Inability to deal with changing requirements.
- Modules that don't fit together.
- Software that's hard to maintain or extend.
- Late discovery of serious projects flaws.
- Poor software quality.
- Unacceptable software performance.
- Team members in each other's way.
- An untrustworthy build and release process.

1.2.3 Root causes of project failure

The same text identifies the root causes of these failures:

- Ad hoc requirements management.
- Ambiguous and imprecise communication.
- Brittle architectures.
- Overwhelming complexity.

- Undetected inconsistencies in requirements, designs, and implementations.
- Insufficient testing.
- Subjective project status assessment.
- Failure to attack risk.
- Uncontrolled change propagation.
- Insufficient automation.

1.2.4 Software development best practices

Finally, the same text identifies these best practices:

- Develop software iteratively.
- Manage requirements.
- Use component-based architectures.
- Visually model software.
- Verify software quality.
- Control changes to software.

1.3 Introduction to OO concepts

What does it mean to be "object-oriented"? The "big three" concepts are encapsulation, polymorphism, and inheritance, but the text "Fundamentals of Object-Oriented Design in UML" [12] specifies that the following criteria are necessary for a language to be considered object oriented.

1. Encapsulation – the grouping of related ideas into unit. Encapsulating attributes and behaviors.
2. Inheritance – a class can inherit its behavior from a superclass (parent class) that it extends.
3. Polymorphism – literally means "many forms".
4. Information/implementation hiding – the use of encapsulation to keep implementation details from being externally visible.
5. State retention – the set of values an object holds.
6. Oject identity – an object can be identified and treated as a distinct entity.
7. Message passing – the ability to send messages from one object to another.
8. Classes – the templates/blueprints from which objects are created.
9. Genericity – the construction of a class so that one or more of the classes it uses internally is supplied only at run time.
10. Test here
11. More testing here

1.3.1 Encapsulation

The grouping of related items into one unit.

- One of the basic concepts of OO.
- Attributes and behaviors are encapsulated to create objects.
- OO modeling is close to how we perceive the world.

- Implementation details are hidden from the outside world. We all know how to use a phone, few of us care how it works.
- The packaging of operations and attributes representing state into an object type so that state is accessible or modifiable only through the objects' interface
- Encapsulation lets builders of objects reuse already-existing objects, and if those objects have already been well-tested, much larger and more complex systems can be created.

1.3.2 Inheritance

- A subclass is derived from a superclass. An `Employee` is a `Person`.
- The subclass inherits the attributes and behavior of the superclass.
- The subclass can override the behavior of the superclass.
- Notice the use of the "Is-A" phrase to describe inheritance.
- Inheritance promotes re-use.

1.3.3 Polymorphism

- Literally means "many forms".
- A method can have many different forms of behavior.
- Commonly used between a set of classes that have a common superclass.
- The sender of a message does not have to know the type/class of the receiver.
- A single operation or attribute may be defined upon more than one class and may take on different implementations in each of those classes.
- An attribute may point to different objects at different times.

1.3.4 Abstraction with objects

- Abstraction: the act of identifying software artifacts to model the problem domain.
- Classes are abstracted from concepts.
- This is the first step in identifying the classes that will be used in your applications.
- Better to have too many classes than too few. (?)
- When in doubt, make it a class. (?)

1.3.5 Message passing

- Objects communicate by sending messages.
- Messages convey some form of information.
- An object requests another object to carry out an activity by sending it a message.
- Most messages pass arguments back and forth.
- Meilir Page-Jones defines three types of messages[12]:
    1. Informative – send information for the object to update itself.
    2. Interrogative – ask an object to reveal some information about itself
    3. Imperative – take some action on itself, or another object
- Grady Booch defines four types of messages[4]:
    1. Synchronous – receiving object starts only when it receives a message from a sender, and it is ready.
    2. Balking – sending object gives up on the message if the receiving object is not ready to accept it.
    3. Timeout – sending object waits only for a certain time period for the receiving object to be ready to accept the message.
    4. Asynchronous – sender can send a message to a receiver regardless of whether the receiver is ready to receive it.

1.4 UML summary

- Unified Modeling Language – UML.
- A modeling language, not a method.
- Provides a graphical representation that allows developers and architects to model a software system before the system is ever built.
- Analogy – an architect creating a blueprint before a house or office building is ever built.
- The UML does not specify a methodology/process. Therefore, saying "We use the UML methodology is incorrect."

A few URLs for reference:

- http://www.omg.org
- http://www.rational.com/uml/index.jsp
- UML Distilled – http://www.awl.com/cseng/titles/0-201-32563-2

1.4.1 Standard diagrams

The UML defines nine standard diagrams:

- Use Case
- Class
- Interaction
    1. Sequence
    2. Collaboration
- Package
- State
- Activity
- Component
- Deployment

Note that the UML can be used to model other processes besides software development.

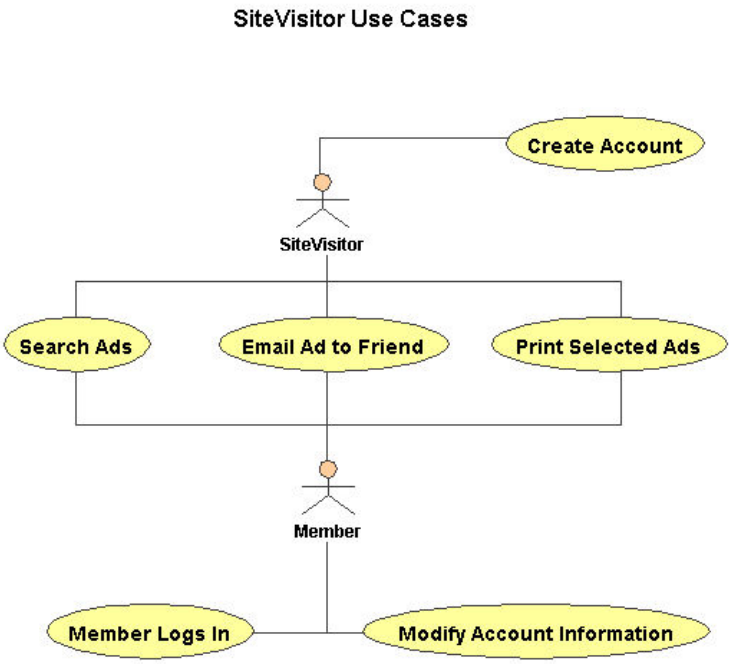

Figure 1.1: A sample UML Use Case Diagram.

Use Case diagram

A typical interaction between a user and a computer system.

Class diagram

- Describes the types of objects in the system, and the static relationships between them.
- Two main kinds of static relationships:
  - Associations – Has-A
  - Subtypes – Is-A

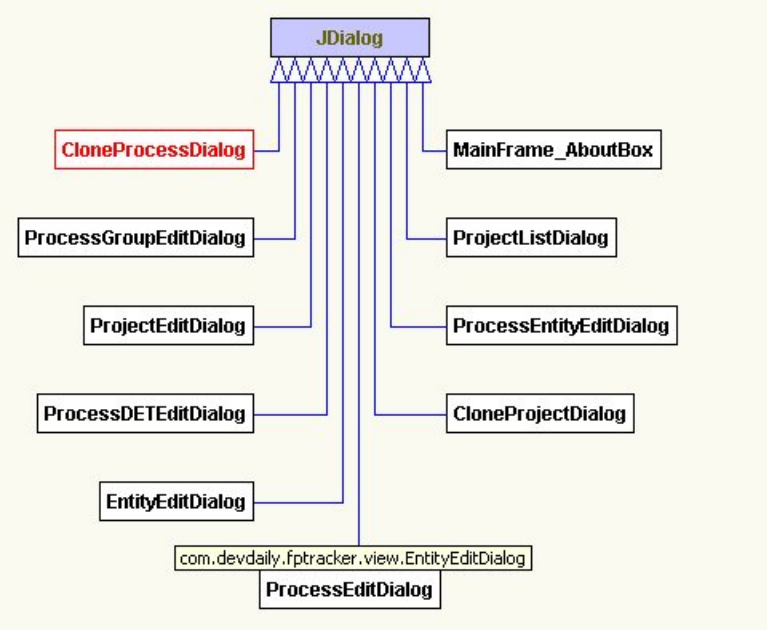

Figure 1.2: A high level class diagram showing the relationships between classes.

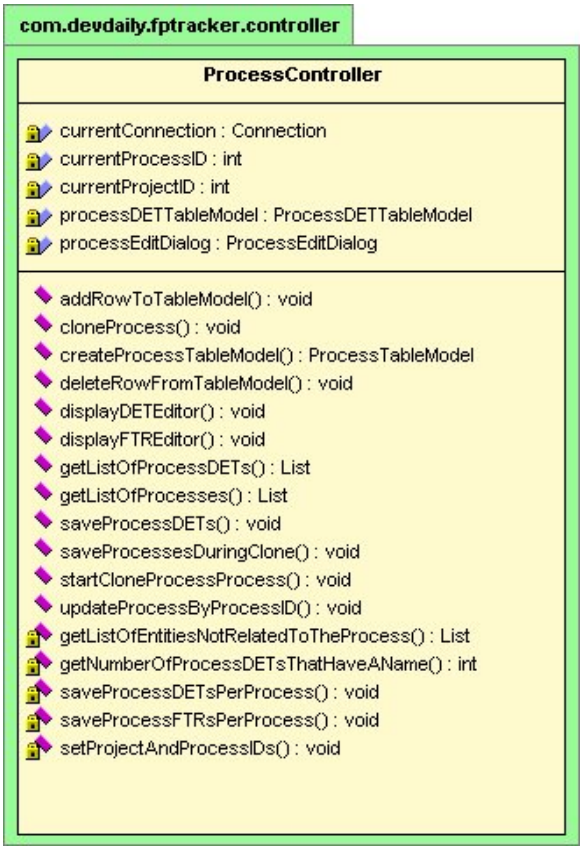

Figure 1.3: A class diagram showing the detailed attributes and behaviors of a class.

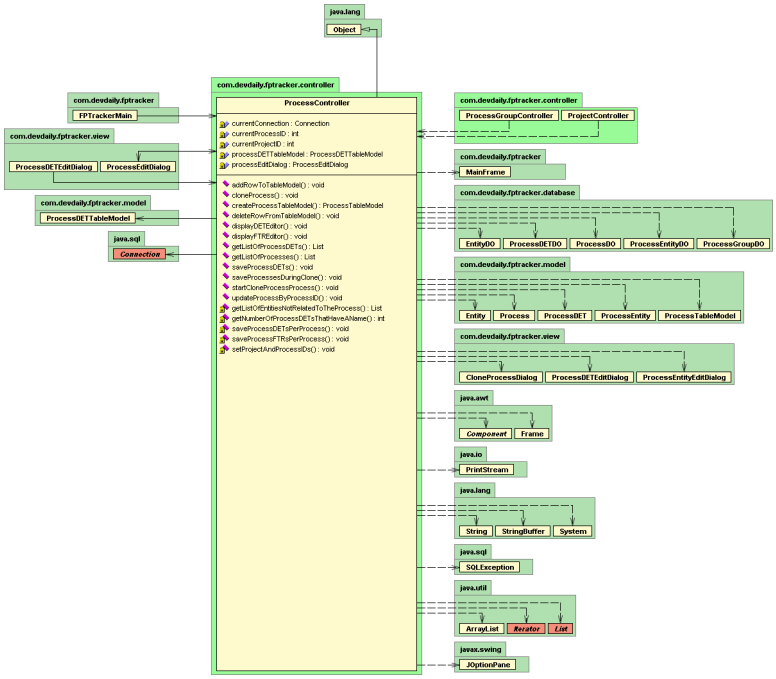

Figure 1.4: A class diagram showing relationships to classes in other packages

Sequence diagram

- Sequence diagrams follow the flow of entire use cases (emphasis on time ordering).
- One sequence diagram for the basic course and alternative courses for each of your use cases.

Collaboration diagram

- Shows how critical objects collaborate within a use case.
- Similar to sequence diagrams.

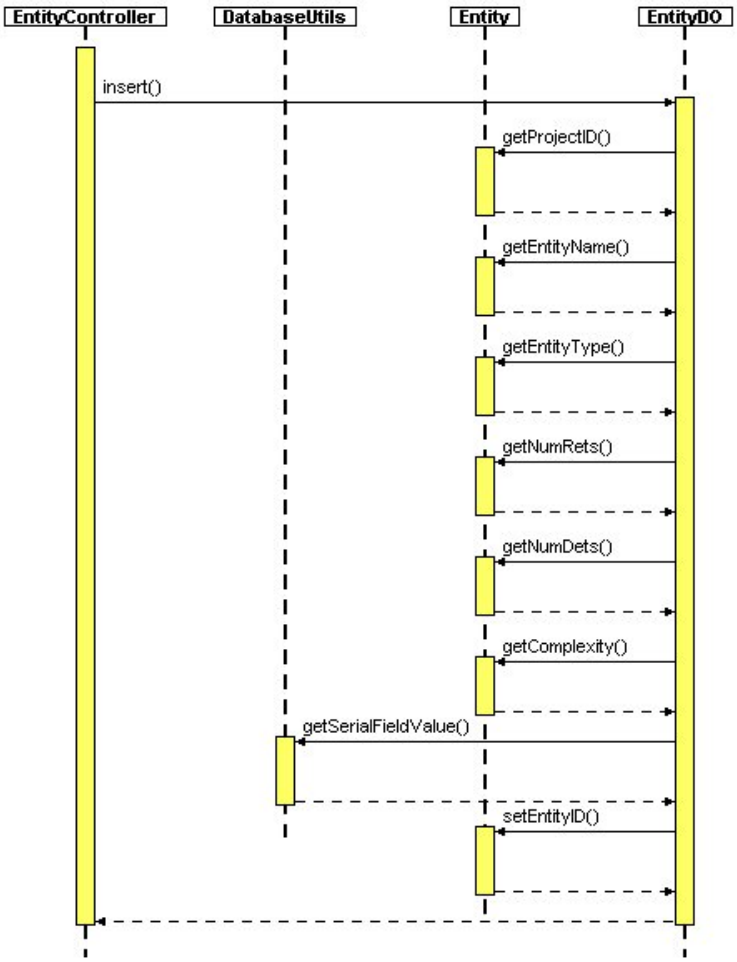

Figure 1.5: A sequence diagram follows the flow of an entire use case.

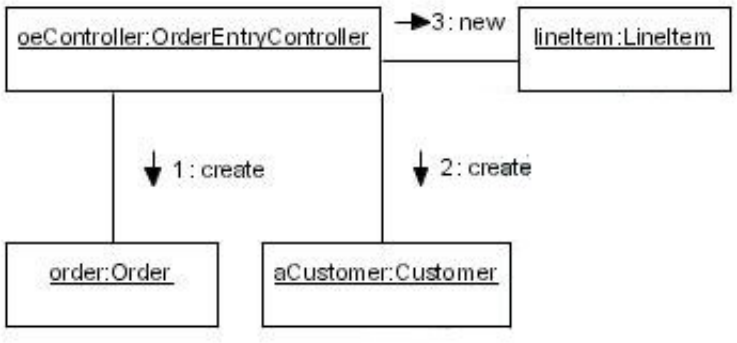

Figure 1.6: A collaboration diagram shows how important objects collaborate within a use case.

- Focus on key transactions.
- Sequence diagrams follow the flow of entire use cases (emphasis on time ordering).
- Collaboration diagrams add extra detail related to timing of messages.

Package diagram

- Classes are arranged into logically-ordered packages.
- Package diagrams show relationships and dependencies between packages.
- Package diagrams are vital for large projects.

State diagram

- Captures the lifecycle of one or more objects.

Activity diagram

- Advanced flowcharts.
- Swimlanes let you organize a set of activities according to who is performing them.

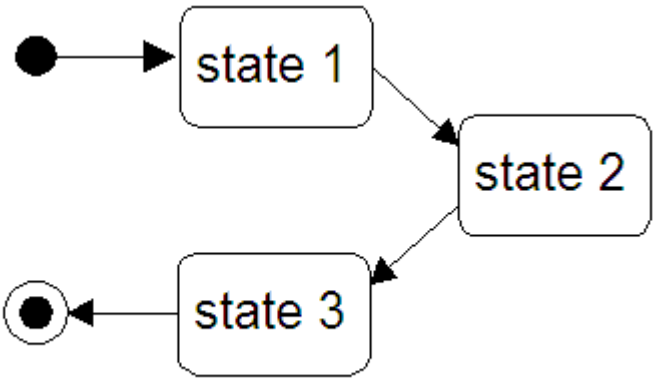

Figure 1.7: A state diagram captures the lifecycle of one or more objects.

Component diagram

- The implementation view of a system.
- Displays the organization and dependencies between software components.

Deployment diagram

- The environment view of a system.
- Shows the physical relationships among software and hardware components.
- Each node represents some computational unit – usually a piece of hardware.
- Connections show communication paths.
- In practice probably not used very much, though most projects have a drawing that looks something like these.

1.5 Object Oriented Software Development

- Larger processes/methodologies:
  - Rational Unified Process (RUP)
  - Object-Oriented Software Process (OOSP)
  - OPEN Process
- Lightweight/agile processes:
  - XP (Extreme Programming)
  - Cockburn's Crystal Family
  - Open Source
  - Highsmith's Adaptive Software Development
  - Scrum
  - Coad's Feature Driven Development
  - DSDM (Dynamic System Development Method)

Larger/heavier processes are typically built around the Unified Modeling Language, or UML.

1.5.1 Why have a process?

- Create a product that users want.
- Make the process manageable and predictable.
- Capability Maturity Model (CMM) from the Carnegie-Mellon University Software Engineering Institute defines five levels of "maturity".
- Traditional development process: Waterfall (Analysis, Design, Code, Test).
- Spiral process (Risk Analysis/Systems Analysis/User Feedback, Design, Code, Test/User Feedback).
- Objectory - defines the four project phases of Inception, Elaboration, Construction, Transition.
  - Completion of each Objectory phase marks a major milestione.
  - Objectory uses iteration to make complex projects possible.

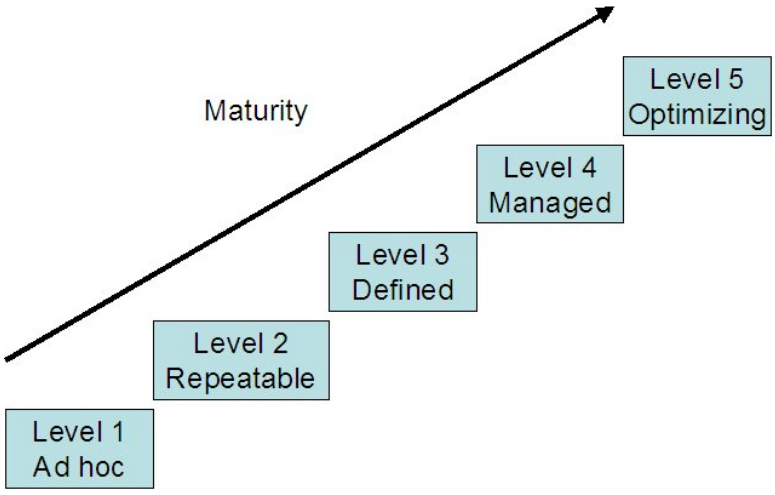

Figure 1.8: Steps of the Capability Maturity Model.

- The process defines ways to make iterative projects manageable.
- Model the system before developing it (in the same way an architect models a new facility before building it).

1.6 The Rational Unified Process (RUP)

The Rational Unified Process formally consists of the following steps:

- Inception – a discover phase, where an initial problem statement and functional requirements are created.
- Elaboration – the product vision and architecture are defined, construction cycles are planned.
- Construction – the software is taken from an architectural baseline to the point where it is ready to make the transition to the user community.
- Transition – The software is turned into the hands of the user's community.

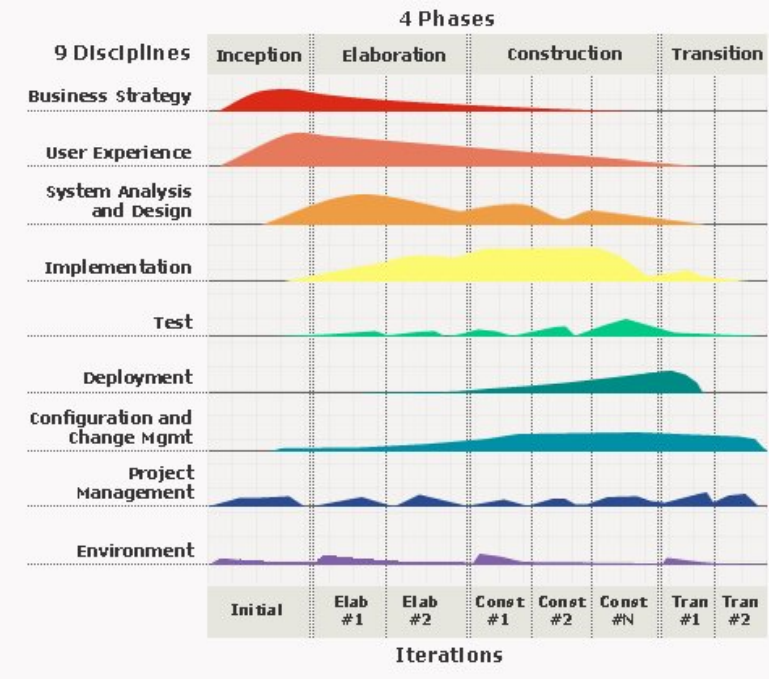

Figure 1.9: A high-level view of the Rational Unified Process. (Image courtesy of Rational Software.)

1.6.1 Inception phase

The "The Rational Unified Process, An Introduction"[9] specifies the following objectives, activities, and artifacts from an inception phase.

Objectives

- Establish project's scope and boundary conditions.
- Determine the critical uses of the system.
- Exhibiting at least one candidate architecture against some of the primary scenarios.
- Estimating the overall cost and schedule for the entire project.
- Estimating potential risks.

Activities

- Formulate the scope of the project.
- Plan and prepare the business case, including alternatives for risk management, staffing, project plan, and trade-offs between cost, schedule, and profitability.
- Develop a candidate architecture.

Artifacts

- A vision document.
- A use-case model survey.
- An initial project glossary.
- An initial business case.
- An initial risk assessment.
- A project plan.
- Other possible items:
  - An initial use-case model.
  - An initial domain model.
  - One or more prototypes.

1.6.2 Elaboration

The "The Rational Unified Process, An Introduction"[9], specifies the following purpose, objectives, activities, and outcome from an elaboration phase.

Purpose

- Analyze the problem domain.
- Establish a sound architecturla foundation.
- Develop the project plan.
- Eliminate the highest-risk elements.

Objectives

- Define, validate, and baseline the architecture.
- Baseline the vision.
- Baseline a plan for the construction phase.
- Demonstrate that the baseline architecture will support the vision for a reasonable cost in a reasonable time, or not.

Activities

- The vision is elaborated.
- The process, infrastructure, and development environment are elaborated.
- Processes, tools, and automation support are put into place.
- The architecture is elaborated and the components are selected.

Outcome/Deliverables

- An 80% complete use-case model has been developed.
- Supplementary requirements are documented.
- A software architecture description is created.
- An executable architectural prototype is created.

- A revised risk list and business case are created.
- A development plan for the overall project is created.
- An updated development case is created, specifying the process to be used.

Other artifacts

- Construction plan.
- Software prototypes.
- Risk identification and management plan.
- A test plan.
- A data dictionary.
- A preliminary user manual.

1.6.3 Construction phase

The "The Rational Unified Process, An Introduction"[9], specifies the following objectives, activities, and deliverables from a construction phase.

Objectives

- Minimize development costs by optimizing resources and avoiding unnecessary scrap and rework.
- Achieving adequate quality as rapidly as practical.
- Achieving useful versions as rapidly as practical.

Activities

- Resource management, resource control, process optimization.
- Complete component development and testing.
- Assessment of product releases against acceptance criteria.

Deliverables

- The software product integrated on the adequate platforms.
- User manuals.
- A description of the current release.

1.6.4 Transition

The text "The Rational Unified Process, An Introduction"[9], specifies the following purpose, objectives and activities from a transition phase.

Purpose

- Transition the software to the user community.
  - Beta testing.
  - Parallel operation with any existing legacy system.
  - Conversions of operational databases.
  - Training of users.
  - Product rollout.

Objectives

- Achieving user self-supportability.
- Achieving stakeholder buy-in that the deployed product is complete and consistent with the evaluation criteria of the vision.
- Achieving final product baseline as rapidly and cost-effectively as possible.

Activities

- Deployment-specific engineering:
  - Cutover
  - Commercial packaging and production
  - Sales rollout
  - Field personnel training
- Tuning activities, bug-fixing, and enhancement for performance and usability.
- Assessing the deployment baseline against the vision and acceptance criteria.

1.7 A sample process

The next several sections provide an outline of a sample object-oriented software development process. Specifically, this process is based on the text "Use Case Driven Object Modeling with UML, A Practical Approach" by Rosenberg and Scott[4]. This is a real process, based on the theory outlined in the Rational Unified Process.

1.7.1 Domain modeling

First guess at domain objects.

What is a class?

- A **class** is a template for creating objects.
- A class defines the attributes and behaviors an object will have.
- An **object** is a specific instance of a class.
- If an object has no attributes, is it really a valid object?

Attributes

- A data variable with object scope.
- Examples: book attributes: title, author, publisher, ISBN
- The value of an object's attributes define its state.
- Attributes should not be accessible to entities outside the object.
- If an object has no attributes, is it really a valid object?

Behaviors

- Method: a function with object scope.
- Methods can operate on that object's attributes.
- Defines the objects behaviors – how it does what it does.
- Methods define the objects responsibilities.
- If an object has no methods, is it really a valid object?

Discover classes

- Work outward from data requirements to build a static model.
- Jump start with **grammatical inspection**: Make a quick pass through the available material, making lists of the nouns, verbs, and possessive phrases.
- Nouns become classes.
- Noun phrases becomes class attributes.
- Verbs become operations (behaviors) and associations.
- Possessive phrases may indicate that nouns should be attributes rather than objects.
- Create this list of "class candidates".
- Best sources of classes: high-level problem statement, lower-level requirements, expert knowledge of the problem space.
- Go through the candidate classes and eliminate the items that are unnecessary (redundant or irrelevant) or incorrect (too vague, represent concepts outside the model).

Where else do classes come from?

- Tangible things – cars, telemetry data, pressure sensors
- Roles – mother, teacher, politician, manager
- Events – interrupt, request
- Interactions – loan, meeting, intersection
- People – humans who carry out some function
- Places – areas set aside for people or things
- Things – physical objects, devices
- Organizations – formally organized collections of people
- Concepts – principles or ideas that are not tangible

Build generalization relationships

- Generalization – one class is a refinement of another class.
- Define Is-a relationships.
- Break into (1) superclass/parent, and (2) subclass/child.
- Child inherits attributes and behaviors of the parent.
- Sometimes discover classes "ahead of schedule".

Build associations between classes

- ***Association***: A static relationship between two classes.
- Show dependencies between classes but not between actions.
- Should be a true statement about the problem space, independent of time (i.e., static).
- Build the list of candidate associations from the list of verbs and verb phrases and knowledge of the problem domain.

Examples:

- Order ***generates*** Trade.
- Portfolio ***places*** Orders.
- Posting ***involves*** GLAccount.
- Trade ***generates*** TradeLot.
- Some associations are one-to-one, some are one-to-many. These are referred to as multiplicities.
- Don't worry about being more specific about numbers of one-to-many associations at this time.
- ***Aggregation***: an association in which one class is made up of other classes. "Has-a" or "part-of" relationships.

Mine legacy documentation for domain classes

- Relational database tables are an excellent source of domain classes.
- ***Helper classes***: contain attributes and operations that are relevant to more significant classes.

Wrapping up domain modeling

Continue to iterate and refine

- Draw an analysis-level class diagram
- The user's wants and needs are the reason your project team exists.
- Establish a time budget for building your initial domain model.
- The diagram you draw during domain modeling is just a skeleton of your object model.

Three key principles

- Work inward from user requirements.
- Work outward from data requirements.
- Drill down from high-level models to detailed design.

1.7.2 Use case modeling

Actors

- An actor is anyone or anything that must interact with the system.
- An actor is not part of the system.
- Actors are typically found in the problem statement, and by conversations with customers and domain experts.
- Actors are drawn as a stick figures.
- When dealing with actors it is important to think about *roles*.

Questions to help identify actors

- Who is interested in a certain requirement?
- Who installs the system?
- Who starts and stops the system?
- Where in the organization is the system used?
- Who will benefit from use of the system?
- Who will supply the system with this information, use this information, and remove this information?
- Who will support and maintain the system?
- Does the system use an external resource?
- Does one person play several different roles?
- Do several people play the same role?
- Does the system interact with a legacy system?
- What other systems use this system?
- Who gets information from this system?

Use Cases Use Case – a sequence of actions that an actor performs within a system to achieve a particular goal. The purpose of this stage is to capture user requirements of the new system using use case diagrams.

- Definition: A sequence of actions that an actor performs within a system to achieve a particular goal.
- A use case describes one or more courses through a user operation. The basic course must always be present; alternate courses are optional.
- Basic course – the main start-to-finish path the user will follow under normal circumstances.
- Alternate course – infrequently used path, an exception, or an error condition.
- *Stated from the perspective of the user* as a present-tense verb phrase in an active voice (AdmitPatient, Do Trade Entry, Generate Reports).
- Describes one aspect of usage of the system without presuming any specific design or implementation.
- Ask "what happens?"
- "Then what happens?"
- Be relentless.
- All required system functionality should be described in the use cases.
- ***Actor*** - represents a role a user can play with regard to a system.
- A user can serve as more than one type of actor.
- Use cases appear as ovals, generally in the middle of a use case diagram.
- Analysis level and design level use cases.
- Should be able to write a solid paragraph or two about a design-level use case.
- Use cases should have strong correlations with material in the user manual for the system (write the manual, then write the code). (Write the manual as though the system already exists.)
- Use *rapid prototyping* as frequently as possible.

- If you're reengineering an existing legacy system, work from the user manual backward.
- A use case captures some user-visible function.
- A use case achieves a discrete goal for the user.
- A use case may be large or small.
- Use cases model a dialogue between an actor and the system.

Questions to help identify use cases[11]

- What are the tasks of each actor?
- Will any actor create, store, change, remove, or read information in the system?
- What use case will create, store, change, remove, or read this information?
- Will any actor need to inform the system about sudden, external changes?
- Does any actor need to be informed about certain occurrences in the system?
- What use cases will support and maintain the system?
- Can all functional requirements be performed by the use cases?

Use case diagrams

- Use case diagrams identify use cases and actors of the system.
- Use case-actor relationships are shown with association lines.
- Use cases appear as ovals, generally in the middle of a diagram.
- Use cases appear at various levels of detail; two such levels are analysis-level and design-level.

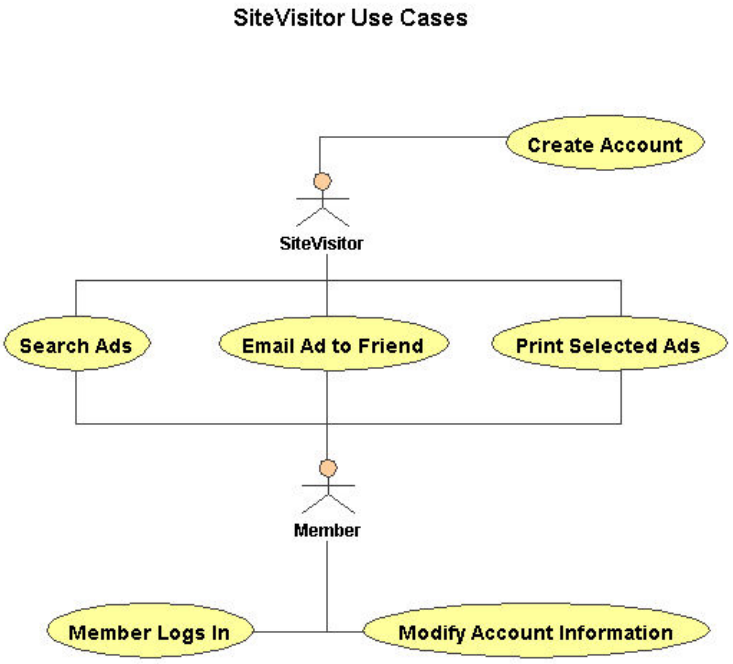

Figure 1.10: A sample use case diagram.

Use case diagrams – include and extend

- One use case can use or extend another use case.
- The secondary (sub-level) use case is not associated directly with an actor.
- Stereotypes
- `<<includes>>` – use/include a piece of behavior that is similar across more than one use case.
- `<<extends>>` – one use case is similar to another, but does more.

Wrapping Up Use Case Modeling Feel comfortable when you've achieved the following goals:

1. Your use cases account for all of the desired functionality of the system.
2. You have clear and concise descriptions of the basic course of action, with appropriate alternate courses of action.
3. You have factored out common scenarios.

Milestone 1: Requirements Review

1.7.3 Robustness analysis

Robustness analysis involves:

1. analyzing the text of each of your use cases,
2. identify a first-guess set of objects that will participate in the use case, then
3. classify these objects into
   (a) Boundary objects
   (b) Entity objects
   (c) Control objects
4. This has parallels in the Model/View/Controller paradigm.

Definitions

- **Boundary objects** - Actors use these to communicate with the system ("View").
- **Entity objects** - Usually objects from the problem domain ("Model").
- **Control objects** - Serve as the "glue" between Boundary and Entity objects ("Controller").

Key roles of robustness analysis

- Sanity check - make sure your use case text is correct.
- Completeness check - make sure your use cases address all the necessary alternate courses of action.
- Ongoing discovery of objects - you may have missed some objects during domain modeling.

Closer look at object types

Boundary objects

- "View".
- Objects that the Actors will be interacting with.
- Windows, screens, dialogs, menus.
- Get many from prototypes.

Entity objects

- "Model".
- Often map to database tables and files.
- Many come from the domain model.
- Simpler and more generic – easier to reuse in other projects.

Control objects

- "Controller".
- Embody much of the application logic.
- Where you capture business rules and application logic.
- Not necessarily meant to endure as stand-alone classes as you proceed.
- Sometimes serve as placeholders to make sure you don't forget any functionality and system behavior required by your uses cases.

Performing robustness analysis

- Actors can talk only to Boundary objects.
- Boundary objects can talk only to Controllers and Actors.
- Entity objects can only talk to Controllers.
- Controllers can talk to both Boundary objects and Controllers, but not to Actors.
- Update your static model.

Milestone 2: Preliminary Design Review

1.7.4 Interaction modeling

Introduction

Current state:

- Uncovered most problem space objects and assigned some attributes to them.
- Defined some static relationships between these objects.
- Defined a few dynamic relationships on robustness diagrams.

Interaction modeling is the phase in which you build the threads that weave your objects together and enable you to see how they will perform useful behavior. One of the primary tools of this task is creating ***sequence diagrams***.

Objectives

Upon completion of this section, students will be able to:

- Define the goals of interaction modeling.
- Create sequence diagrams.
- Put behavioral methods on your classes.
- Update your static model.

Goals of Interaction Modeling

- Allocate behavior among entity, boundary, and control objects.
- Show detailed interactions that occur over time among objects.
- Finalize the distribution of operations among classes.

Sequence Diagrams

- Represent the major work product of our design.
- Draw one sequence diagram that encompasses the basic course and all alternative courses within each of your use cases. *One sequence diagram per use case.*
- These results form the core of your dynamic model.

Four Sequence Diagram Elements

- The text for the course of action of the use case.
- Objects.
- Messages.
- Methods (operations, behaviors).

Getting Started

Four steps to creating diagrams:

- Copy the use case text to the left margin of the sequence diagram.
- Add the entity objects.
- Add the boundary objects.
- Work through the controllers, one at a time. Determine how to allocate behavior between the collaborating objects.

Putting Methods on Classes

- This is the essence of interaction modeling.
- It's also hard.
- A cow needs milking. Does the cow object milk itself, or does the Milk object "de-cow" itself?
- Convert controllers from robustness diagrams to sets of methods and messages that embody the desired behavior.
- An object should have a single "personality". Avoid schizophrenic objects.
- If you have objects with split personalities you should use aggregation.
- Use CRC cards to help.
- Behavior allocation is of critical importance.
- Don't show message parameters on your sequence diagrams.

Which Methods Belong With Objects

- Reusability - the more general, the more reusable. Does this method make the class more or less reusable.
- Applicability - is there a good fit between the object and method?
- Complexity - is it easier to build a method in another object?
- Implementation knowledge - does the implementation of the behavior depend on details internal to the associated method?

Completing Interaction Modeling

- Drawn all needed sequence diagrams.
- Updated your static model.
- Last stop before you start coding; Critical Design Review is essential.

1.7.5 Collaboration and State Modeling

Introduction

- Model additional aspects of your system.
- Most useful in real-time system design.
- Collaboration diagrams are similar to sequence diagrams.

Collaboration diagrams

- Shows how critical objects collaborate within a use case.
- Collaboration diagrams are similar to sequence diagrams.
  - Collaboration diagrams focus on key transactions.
  - Sequence diagrams follow the flow of entire use cases (emphasis on time ordering).
  - Collaboration diagrams add extra detail related to timing of messages.

State diagrams

- Captures the lifecycle of one or more objects.
- Expressed in terms of:
  - Different states objects can assume
  - Events that cause changes in state

Basic elements:

- Initial state – hollow circle containing a black dot.
- Each additional state – rectangle with rounded corners.
- Three standard events:
  - Entry
  - Exit
  - Do
- Transition – an arrow between two states.

How many state diagrams are needed?

- Every object has a state machine.
  - Object is created.
  - Sends messages.
  - Receives messages.
  - It is destroyed.
- In reality, most state machines are boring, so don't waste time drawing them.
  - Don't diagram an object with two states, On and Off.
- Readability is important.

Activity diagrams

- Remarkably similar to flowcharts.
- Swimlanes – group a set of activities according to who is performing them.
- A good way to understand/model business processes.

1.7.6 Addressing Requirements

Introduction

This section shows how to trace the results of your analysis and design work back to your user requirements.

Objectives

Upon completion of this section, students will be able to:

- Define a requirement
- Describe the nature of requirements, use cases, and functions

What is a Requirement?

- A user-specified criterion that the system must satisfy.
- Requirements define the bahvior and functionality of a proposed system.
- Usually expressed as sentences that include the word *shall* or *must*.

Types of Requirements

- Functional - "The system shall automatically generate postings to the general ledger".
- Data - "The system ... international currencies ...".
- Performance - "The system must ... in XX seconds".
- Capacity - "Up to 10,000 transactions per day".
- Test - "Stress testing shall ... XX users ... YY computers ...".

Use Cases and Requirements

- A use case describes a unit of behavior.
- A *requirement* describes a law that governs behavior.
- Several types of requirements: functional, performance, and constraints.
- A use case can satisfy one or more functional requirements.

- A functional requirement may be satisfied by one or more use cases.
- Requirements are requirements, use cases are use cases. Requirements are not use cases.

Requirements Traceability

- Make a list of the system requirements.
- Write the user manual for the system in the form of use cases.
- Iterate with your customers until you have closure of items 1 and 2.
- Make sure you can trace every piece of your design to at least one user requirement.
- Make sure you can trace every requirement to the point at which its satisfied within your design.
- Trace your design back to your requirements as you review the design during your critical design review.

1.7.7 Survey of Design Patterns

- Repeating patterns - "Her garden is like mine, except that in mine I use astilbe."
- Recurring solutions to design problems you see over and over.
- A set of rules describing how to accomplish certain tasks in the realm of software development.
- Made famous by the text "Design Patterns", by Gamma, Helm, et al.
- A list of some of the most well-known design patterns:

Factory	Abstract Factory
Singleton	Builder
Prototype	Adapter
Bridge	Composite
Decorator	Facade
Flyweight	Proxy
Chain of Responsibility	Command
Interpreter	Iterator
Mediator	Memento
Observer	State
Strategy	Template
Visitor	

Factory pattern example

- The code and diagram that follow demonstrate a small, simple example of the Factory pattern

```
public abstract class Dog {
  public abstract void speak ();
}

public class Poodle extends Dog {
  public void speak() {
    System.out.println("The poodle says \"arf\"");
  }
}

public class SiberianHusky extends Dog {
  public void speak() {
    System.out.println("The husky says \"Whazzup?!!\"");
  }
}

public class Rottweiler extends Dog {
  public void speak() {
    System.out.println("The Rottweiler says (in a very deep voice) \"WOOF!\"");
  }
}
```

```
public class Main {
  public Main() {
    // create a small dog
    Dog dog = DogFactory.getDog("small");
    dog.speak();

    // create a big dog
    dog = DogFactory.getDog("big");
    dog.speak();

    // create a working dog
    dog = DogFactory.getDog("working");
    dog.speak();
  }

  public static void main(String[] args) {
    new Main();
  }
}

public class DogFactory {

  public static Dog getDog(String criteria) {
    if ( criteria.equals("small") )
      return new Poodle();
    else if ( criteria.equals("big") )
      return new Rottweiler();
    else if ( criteria.equals("working") )
      return new SiberianHusky();

    return null;
  }
}
```

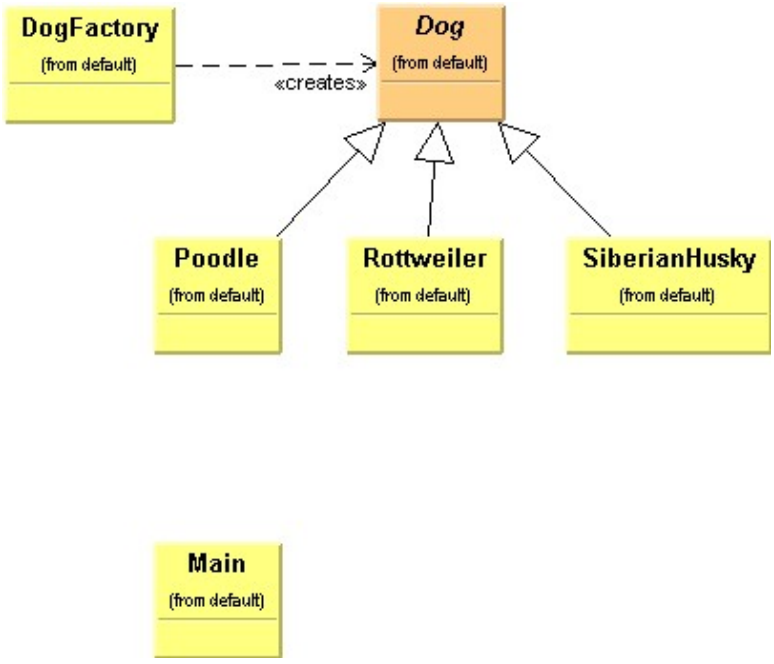

Figure 1.11: A UML use case diagram for the DogFactory.

1.8 Agile Methods

Many developers believe that formal "heavyweight" processes like RUP still have significant shortcomings. According the original Extreme Programming text (the "white book"), at a minimum the following roblems still exist:

- Schedule slips
- Project canceled
- System goes sour
- Defect rate
- Business misunderstood
- Business changes
- False feature rich
- Staff turnover

1.9 The Agile Alliance

A set of principles from the "Agile Alliance" (http://www.agilealliance.org/):

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Business people and developers must work together daily throughout the project.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

- Working software is the primary measure of progress.
- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances agility.
- Simplicity–the art of maximizing the amount of work not done–is essential.
- The best architectures, requirements, and designs emerge from self-organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

1.10 Introduction to Extreme Programming

This is a simple introduction to Extreme Programming, taken largely from the text "eXtreme Programming explained" (the "white book").

1.10.1 Risk: The Basic Problem

Quoting from the white book, "the basic problem of software development is **risk**". Here are some examples:

- Schedule slips
- Project canceled
- System goes sour
- Defect rate
- Business misunderstood
- Business changes
- False feature rich
- Staff turnover

1.10.2 Four Variables

In the XP model there are four control variables in software development:

1. Cost
2. Time
3. Quality
4. Scope

Customers and managers get to pick the values of any three of the variables.

1.10.3 The Cost of Change

Under certain circumstances the exponential rise in the cost of changing software over time can be flattened. If the cost curve can be flattened, old assumptions about the best way to develop software no longer hold true. Several factors make code easy to change, even after years of production:

1. A simple design
2. Automated tests
3. Lots of practice in modifying the design

1.10.4 Four Values

1. Communication
2. Simplicity
3. Feedback
4. Courage

1.10.5 Basic Principles

1. Rapid Feedback
2. Assume simplicity
3. Incremental change
4. Embracing change
5. Quality work

1.10.6 Back to Basics

1. Coding
2. Testing
3. Listening
4. Designing

1.10.7 The Solution

1. The Planning Game
   Business people get to decide:

   - Scope
   - Priority
   - Composition
   - Dates of releases

   Technical people get to decide:

   - Estimates
   - Consequences
   - Process
   - Detailed scheduling

2. Small Releases
3. Metaphor
4. Simple Design
5. Testing
6. Refactoring
7. Pair Programming
8. Collective Ownership
9. Continuous Integration
10. 40-Hour Week
11. On-Site Customer
12. Coding Standards

1.11 OO Summary

This section provides a summary of our Day One activities.

1.11.1 OO Concepts

Software concepts essential to object orientation:

- Encapsulation – the grouping of related ideas into unit. Encapsulating attributes and behaviors.
- Inheritance – a class can inherit its behavior from a superclass (parent class) that it extends.
- Polymorphism – literally means "many forms".
- Information/implementation hiding – the use of encapsulation to keep implementation details from being externally visible.
- State retention – the set of values an object holds.
- Oject identity – an object can be identified and treated as a distinct entity.
- Message passing – the ability to send messages from one object to another.
- Classes – the templates/blueprints from which objects are created.
- Genericity – the construction of a class so that one or more of the classes it uses internally is supplied only at run time.

1.11.2 UML

The UML defines nine standard diagrams:

1. Use Case
2. Class
3. Interaction
   (a) Sequence
   (b) Collaboration

4. Package
5. State
6. Activity
7. Deployment

Rational Unified Process

- Inception – a discover phase, where an initial problem statement and functional requirements are created.
- Elaboration – the product vision and architecture are defined, construction cycles are planned.
- Construction – the software is taken from an architectural baseline to the point where it is ready to make the transition to the user community.
- Transition – The software is turned into the hands of the user's community.

Agile Methods

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Business people and developers must work together daily throughout the project.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- Working software is the primary measure of progress.

- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances agility.
- Simplicity–the art of maximizing the amount of work not done–is essential.
- The best architectures, requirements, and designs emerge from self-organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Design Patterns

- Repeating patterns - "Her garden is like mine, except that in mine I use astilbe."
- Recurring solutions to design problems you see over and over.
- A set of rules describing how to accomplish certain tasks in the realm of software development.
- Made famous by the text "Design Patterns", by Gamma, Helm, et al.

Chapter 2

Day 2: The Java Programming Language

2.1 Introduction

2.1.1 Chapter objectives

- Review the Java design goals.
- Understand the reserved Java keywords.
- Work with Java strings and arrays.
- Create standalone Java applications.
- Create Java classes that use and extend one another.
- Write programs that use and handle exceptions.
- Learn about Java interfaces.
- Understand how to package your Java classes.

2.1.2 Java design goals

- Simple – syntax like C, but easier.
- Secure – compile- and runtime-support for security.
- Distributed – built to run over networks.
- Object-oriented – designed from the ground-up to be object-oriented.
- Robust – strongly typed, memory management, exception handling.
- Portable – "Write Once, Run Anywhere". Runs on any platform with a JVM; Windows, Unix, Linux, Apple, AS/400, cell phones, desktop sets, ...
- Interpreted – Java bytecode is portable.
- Multithreaded – much easier to write multithreaded programs.
- Dynamic – classes are loaded as needed.
- High-performance – just-in-time compilers, advanced memory management makes Java programs faster.

2.1.3 What is Java?

- A very portable object-oriented programming language.
- A large supporting class library that covers many general needs.
- Can creates Applets, Applications, Servlets, JavaServer Pages, and more.
- An open standard – the language specification is publicly available.
- JVM - Java Virtual Machine.

2.1.4 How/where to get Java

- `http://java.sun.com`
- IBM
- A variety of IDE's (Integrated Development Environments)
  - Borland – JBuilder

- IntelliJ – IDEA
- IBM – Visual Age for Java
- Symantec/BEA – Visual Cafe
- Open Source – Netbeans
- More ...

2.2 First Steps with Java

2.2.1 Java Commands and Utilities

- `javac` - the Java compiler
- `java` - the Java bytecode interpreter (JVM)
- `appletviewer` - lets you view applets without a browser
- `jdb` - the Java debugger
- `javadoc` - a utility that lets you generate documentation from your Java source code and the Javadoc comments you place in your source code
- `jar` - Java archive utility (similar to the Unix `tar` command)

2.2.2 A first application

- Assuming you have the JDK downloaded/installed, create a first Java application:

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

- Save the file as `Hello.java`.
- Compile the file to Java bytecode:

```
javac Hello.java
```

- Run the program:

```
java Hello
```

2.2.3 main

- When you run a Java *application*, the system locates and runs the `main` method for that class.
- The `main` method must be `public`, `static`, and `void`.
- The `main` method must accept a single argument of type `String[]`.
- The arguments in the `String` array passed to `main` are program arguments, or command-line arguments.
- An application can have any number of `main` methods, because each class can have one.
- This is good, because each class can have a `main` method that tests it's own code.

2.3 Variables, constants, and keywords

2.3.1 Primitive data types

- Java has built-in "primitive" data types.
- These primitives support integer, floating-point, boolean, and character values.
- The primitive data types of Java are:

boolean	either true or false
char	16-bit Unicode 1.1 character
byte	8-bit integer (signed)
short	16-bit integer (signed)
int	32-bit integer (signed)
long	64-bit integer (signed)
float	32-bit floating-point
double	64-bit floating-point

- Samples:

```
int i = 1;
int age = 38;
float balance = 1590.55;
char a = 'a';
boolean isTrue = true;
```

2.3.2 Literals

- A literal is a vlue that can be assigned to a primitive or string variable, or passed as an argument to a method call.

***boolean* literals**

- `true`
- `false`
- `boolean isTrue = true;`

***char* literals**

- `'\n'` – newline
- `'\r'` – return
- `'\t'` – tab

***Floating-point* literals**

- Expresses a floating-point numerical value.
- `3.1438` – a decimal point
- `4.33E+11` – E or e; scientific notation
- `1.282F` – F or f; 32-bit float
- `1355D` – D or d; 64-bit double

***String* literals**

- A sequence of text enclosed in double quotes.
- `String fourScore = "Four score and seven years ago";`

2.3.3 Constants

- Constants are variables whose value does not change during the life of the object.
- Java does not have a constant keyword.
- In Java you define a constant like this:

```
static final String name = "John Jones";
static final float pi = 3.14159;
```

2.3.4 Reserved keywords

- Reserved Java keywords:

abstract	default	if	private	throw
boolean	do	implements	protected	throws
break	double	import	public	transient
byte	else	instanceof	return	try
case	extends	int	short	void
catch	final	interface	static	volatile
char	finally	long	super	while
class	float	native	switch	
const	for	new	synchronized	
continue	goto	package	this	

Fibonnaci program

- Here is a Fibonnaci program from The Java Programming Language:

```
class Fibonacci {
   public static void main (String args[])
   {
      int lo = 1;
      int hi = 1;
      System.out.println(lo);
      while ( hi < 50 )
      {
         System.out.println(hi);
         hi = lo + hi;
         lo = hi - lo;
      }
  }
}
```

- Type this code into the proper filename.
- Compile and run the program.
- Discuss the results.

2.4 Arrays

- An array is a collection of variables of the same type.
- A variable declared in brackets, [], is an array reference.
- Three steps
  - Declaration – tell the compiler what the name is and the type of its elements.
  - Construction –
  - Initialization

```
int numbers[];          // declaration
numbers = new int[100]; // construction
for (int i=0; i<100; i++)
{
  numbers[i] = i;
}
```

```
String username[];
float[] balance;
```

- Components of an array are accessed by a simple integer index.
- Element indexing begins with the number 0.

```
username[0] = "Fred";
username[1] = "Barney";
```

- The size of an array is easily accessed with the `length` attribute.

```
for (int i=0; i<username.length; i++)
{
    System.out.println("The user's name is: " + username[i]);
}
```

- Indexing past the end of an array throws an exception.
- Multidimensional arrays can be created.

2.5 Strings

2.5.1 String objects

- Java provides a `String` class to deal with sequences of characters.

```
String username = "Fred Flinstone";
```

- The `String` class provides a variety of methods to operate on String objects.
- The `equals()` method is used to compare Strings.
- The `length()` method returns the number of characters in the String.
- The `+` operator is used for String concatenation.
- `String` objects are read-only (also called *immutable*).
- In the example below, the second assignment gives a new value to the object reference `str`, not to the contents of the string:

```
str = "Fred";
str = "Barney"
```

- An array of `char` is not a `String`.

2.5.2 `StringBuffer` class

- The `StringBuffer` class is often used when you need the ability to modify strings in your programs.
- Manipulating a `StringBuffer` can be faster than creating-re-creating `String` objects during heavy text manipulation.
- StringBuffer objects are mutable.
- Useful methods include `append()`, `charAt()`, `indexOf()`, `insert()`, `length()`, and `replace()`.

Exercise

- What is the output of the following class?

```
class StringTest {
   public static void main (String args[]) {
      String str1 = null;
      String str2 = null;
      str1 = "Fred";
      str2 = str1;
      System.out.println("str1: " + str1);
      System.out.println("str2: " + str2);

      str1 = "Barney";
      System.out.println("str1: " + str1);
      System.out.println("str2: " + str2);
   }
}
```

Exercise

- Modify the Hello application so it reads the a user's name from the command line, and writes the user's name as part of the output.
- Here is the source code for the original class:

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

- Assuming that the users name is `Al`, after the changes the output of the program should be:

```
Hello, Al
```

2.6 Comments and Javadoc

2.6.1 Types of comments

- Three different ways to put comments in your Java code:

`//`	define a comment to the end of the current row
`/* ... */`	define a multi-line comment between the two symbols
`/** ...*/`	a "Javadoc" multi-line comment

2.6.2 Javadoc comment tags

- The following special tags in Javadoc comments have predefined purposes.

@see

- Creates a link to other javadoc documentation.
- Name any identifier, but qualify it sufficiently.

```
@see Attr
@see COM.missiondata.web.utils.PageFactory;
```

@param

- Documents a single parameter to a method.
- Have one for each parameter of the method.
- First word is the parameter name, the rest is its description.

```
@param max The maximum number of words to read.
```

@return

- Documents the return value of a method:

```
@return The number of words actually read.
```

@exception

- Documents an exception thrown by the method.
- Should have one for each type of exception the method throws.

```
@exception UnknownName The name is unknown.
@exception IllegalArgumentException
The name is <code>null</code>.
```

@deprecated

- Marks an identifier as being unfit for continued use.
- Code using a deprecated type, constructor, method or field will generate a warning when compiled.

```
\begin{enumerate}
\item
\item @deprecated Do not use this anymore.
\item
\end{enumerate}
```

@author

- Specify the author of the code.

```
@author Alvin Alexander
```

@version

- Specify an arbitrary version.

```
@version 1.11
```

@since

- Specify an arbitrary version specification that denotes when the tagged entity was added to your system

```
@since 2.1
```

2.6.3 A comment example

- A heavily-commented `Attr` class:

```
class Attr
{
    \begin{enumerate}
    \item  The attribute name. */
    \end{enumerate}
 \end{enumerate}
   private String name;

    \begin{enumerate}
    \item  The attribute value. */
    \end{enumerate}
 \end{enumerate}
   private Object value = null;

  /**
   * Creates a new attribute.
   * @see Attr2
   */
   public Attr (String name)
   {
      this.name = name;
   }
}
```

2.6.4 Notes on Usage

- Use the javadoc command to print a separate set of HTML javadoc pages for your classes.

- See http://java.sun.com/j2se/1.4/docs/api/ for an example of javadoc documentation pages for the Java2 Platform.
- Comment skew – comments become out of date as the source code changes over time.
- Some programmers write documentation, others do not.
- Some businesses have technical writers that end up needing write access to your Java source code files.

2.7 Flow control and loops

2.7.1 Introduction

A program consisting of only consecutive statements is immediately useful, but very limited. The ability to control the order in which statements are executed adds enormous value to any program. This lesson covers all the control flow statements that direct the order of execution, except for exceptions.

2.7.2 Objectives

Upon completion of this section, students will be able to:

- Define expression statements and declaration statements.
- Describe the operation of Java's control-flow statements.
- Use Java's `if-else`, `switch`, `while`, `do-while`, and `for` statements.
- Use labels, and labeled `break` and `continue` statements.

2.7.3 Statements and blocks

- Two basic statements – *expression statements*, *declaration statements*.
- Expression statements – such as `i++` –have a semi-colon at the end.

Expressions that can be made into statements

- Assignment: those that contain `=`
- Prefix or postfix forms of `++` and `--`
- Methods calls.
- Object creation statements (new operator).

Declaration statements

- Declare a variable and initialize it to a value.
- Can appear anywhere inside a block.

- Local variables exist only as long as the block containing their code is executing.
- Braces group zero or more statements into a **block**.

2.7.4 if-else

- Basic form of conditional control flow.

```
if (boolean-expression)
   statement1
else
   statement2
```

- Example

```
class IfElse1 {
   public static void main (String[] args) {
   int i = 10;
   if ( i==1 )
      System.out.println( "i == 1" );
   else if ( i==2 )
      System.out.println( "i == 2" );
   else if ( i==3 )
      System.out.println( "i == 3" );
   else
      System.out.println( "don't know what 'i' is" );
   }
}
```

2.7.5 switch

- Evaluates an integer expression in the switch statement.
- Transfers control to an appropriate case label.
- If no match is found, control is transferred to a default label.
- If there is no default label and no other match is found, the switch statement is skipped.

- A break statement is usually used at the end of each case. If a break is not used, control flow *falls through* to the next case.
- All case labels must be constant expressions.
- The value that you are switching on must be `byte`, `short`, `char`, or `int`.
- Example:

```
switch (verbosity)
{
   case BLATHERING:
      System.out.println("blah blah blah ... my name is ...");
   case NORMAL:
      System.out.println("What is your name?");
   case TERSE:
      System.out.println("Yo.");
      break;
   default:
      System.out.println("Hello.");
}
```

2.7.6 while and do-while

- A while loop is executed repeatedly until its boolean expression evaluates to false.
- A while loop will execute zero or more times.
- The boolean-expression can be any expression that returns a `boolean` value.

```
while (boolean-expression)
   statement
```

- A do-while loop executes at least once:

```
do
   statement
while (boolean-expression)
```

2.7.7 for

- Used to loop over a range of values from beginning to end.

```
for (init-expr; boolean-expr; incr-expr)
   statement
```

- Typically used to iterate a variable over a range of values.

```
char ch[] = new char[s.length()];
for (int i=0; i<s.length(); i++)
{
   ch[i] = s.charAt(i);
}
```

2.7.8 Labels

- Statements can be labeled.
- Typically used on blocks and loops.

```
label : statement
```

2.7.9 break

- Used to exit from any block (not just a switch).
- Most often used to break out of a loop.
- An unlabeled break terminates the innermost switch, for, while, or do-while.
- To terminate an outer statement, label the outer statement and use its label name in the break statement.

2.7.10 continue

- Skips to the end of a loop's body and evaluates the boolean expression that controls the loop.

- Has meaning only inside loops: while, do-while, and for.
- A labeled continue will break out of any inner loops on its way to the next iteration of the named loop.

2.7.11 return

- Terminates execution of a method and returns to the invoker.
- If the method has a return type, the return must include an expression of a type that could be assigned to the return type.

```
public static double absolute (double val)
{
   if ( val<0 )
      return -val;
   else
      return val;
}
```

2.7.12 No `goto` Statement

- No goto construct to transfer control to an arbitrary statement in a method.
- Primary uses of `goto` in other languages:
  - Controlling outer loops from within nested loops. Java provides labeled break and continue.
  - Skipping the rest of a block of code that is not in a loop when an answer or error is found. Use a labeled break.
  - Executing cleanup code before a method or block of code exits. Use a labeled break, or the finally construct of the try statement.

2.8 Classes and objects

2.8.1 Introduction

- Class – the fundamental unit of programming in Java.
- Classes contain the attributes and behaviors of the objects you will create.
- Classes define how an object should be created, how it should be destroyed, and how it should behave during it's existence.

2.8.2 Objectives

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

- Define the difference between a Java class and an object.
- Create a simple Java class.
- Create constructors for your classes.
- Define the behavior of your classes.
- Be able to describe Java's garbage collection process.
- Declare and initialize instance variables.
- Access data members and methods of an instance.
- Discuss nested classes and Interfaces.
- Write your own complete Java classes.

2.8.3 A Simple Class

- Each object is an instance of a class.
- The basics of a class are it's fields and methods (or, attributes and behaviors).

```
class Body {
   public long idNum;
   public String nameFor;
   public Body orbits;
```

```
    public static long nextID = 0;
}
```

- First declare the name of the class:

```
Body mercury;
Body earth;
```

- `mercury` and `earth` are references to objects of type `Body`.
- During it's existence, `mercury` may refer to any number of `Body` objects.
- Note - this version of `Body` is poorly designed.

2.8.4 Fields

- A class's variables are called *fields* (or *attributes*).
- Every Body object has its own specific instances of these fields, except for `nextID`.
- Changing the orbits of one object will not affect the orbits of others.

2.8.5 Access Control and Inheritance

- We declared many fields of Body to be public; this is not always a good design idea.
- Four possible access control modifiers:
  - `private` – members declared private are accessible only in the class itself.
  - `protected` – accessible in the class itself, and are accessible to, and inheritable by, code in the same package, and code in subclasses.
  - `public` – accessible anywhere the class is accessible, and inherited by all subclasses.
  - `package` – members declared with no access modifier are accessible in the class itself and are accessible to, and inheritable by, code in the same package.

2.8.6 Creating Objects

- Objects are created using the `new` construct:

```
Body sun = new Body();

Body earth;
earth = new Body();
```

- This example declare two references (sun, earth) that can refer to objects of type Body.
- Using `new` is the most common way to create objects.
- Java runtime (a) allocates enough space to store the fields of the object, (b) initializes the object, and (c) returns a reference to the new object.
- If the system can't find enough free space, it runs the garbage collector.
- If there is not enough free space available, new will throw an OutOfMemoryError exception.

2.8.7 Constructors

- Constructors have the same name as the class they initialize.

```
Body ()
{
   idNum = nextID++;
}
```

- Move the responsibility for `idNum` inside the `Body` class.

```
Body sun = new Body();    // idNum is 0
Body earth = new Body();  // idNum is 1
```

- Constructors can take zero or more parameters.
- Constructors have no return type.
- Constructors are invoked after the instance variables of the new object have been assigned their default initial values, and after their explicit initializers are executed.

Constructor example

- The following three classes demonstrate how constructors are called for classes and superclasses.

```
public class ParentClass {
  public ParentClass() {
    System.out.println( "ParentClass constructor was called" );
  }
}

public class ChildClass extends ParentClass {
  public ChildClass() {
    System.out.println( "ChildClass constructor was called" );
  }
}

public class Main {
  public static void main(String[] args) {
    ChildClass cc = new ChildClass();
  }
}
```

2.8.8 Methods

- Here is a sample method for the `Body` class. The method name is `toString`.

```
  public String toString()
  {
      String desc = idNum + " (" + name + ")";
      if orbits != null)
      {
          desc += " orbits " + orbits.toString();
      }
      return desc;
  }
```

- Returns type `String`.

- Methods contain the code that understands and manipulates an object's state.
- Methods are invoked as operations on an object using the `.` (dot) operator.
- Each method takes a specific number of parameters.
- Each parameter has a specified type - primitive or object.
- Methods also have a return type (or `void`).

Parameter values

- All parameter methods are "pass by value".
- Values of a parameter are copies of the values in the invoking method.
- When the parameter is an object, the object *reference* is passed by value.
- End result: *primitives* cannot be modified in methods, *objects* can.

Using methods to control access

- If data fields (attributes) are public, programmers can change them.
- Generally want to hide the data from programmers that will use the class.
- If programmers can access a class's fields directly, you have no control over what values they can assign.
- In the `Body` example, `nextID`should be `private`.
- If necessary, you should provide *get* methods that allow programmers to determine the current field value, and *set* methods to modify the value. These are called *accessors* (get) and *mutators* (set).

2.8.9 this

- Typically use `this` only when needed.
- Most commonly used as a way to pass a reference to the current object as a parameter to other methods.

- Often used in a case like this:

```
class Pizza
{
    String topping;
    Pizza (String topping)
    {
        this.topping = topping;
    }
}
```

2.8.10 Overloading methods

- Each method has a *signature*.
- The signature is (a) the name together with (b) the number and (c) types of parameters.
- Two methods in the same class can have the same name if they have different numbers or types of parameters.
- This capability is called *overloading*.
- The compiler compares the number and types of parameters to find the method that matches the signature.
- The signature does not include the return type or list of thrown exceptions, and you cannot overload based on these factors.

```
public void aMethod(String s) {}
public void aMethod() {}
public void aMethod(int i, String s) {}
public void aMethod(String s, int i) {}
```

2.8.11 Overriding methods

- You override the signature of a method from a superclass.
- Overriding methods must have argument lists with the identical type and order as the superclass.
- When you override a method of a superclass you should honor the intended behavior of the method.

2.8.12 Static members

- A static member is a member that exists only once per class, as opposed to once per object.
- When you declare a *field* to be static, that means that this field exists only once per class, as opposed to once for each object (such as the `nextID` field of the `Body` object).
- A static method is invoked on behalf of the entire class.
- A static method can access only static variables and static methods of the class.

```
prime = Primes.nextPrime();
id = Vehicle.currentID();
```

2.8.13 Initialization Blocks

- A class can have initialization blocks to set up fields or other necessary states.
- Typically these blocks are static.
- Most useful when simple initialization clauses on a field declaration need a little help.

```
class Primes {
   protected static int[] knownPrimes = new int[4];
   static
   {
      knownPrimes[0] = 2;
      for (int i=1; i<knownPrimes.length; i++)
      {
         knownPrimes[i] = nextPrime();
      }
   }
}
```

- You can also have non-static initialization blocks.

2.8.14 Garbage collection and finalize

- Java performs garbage collection for you and eliminates the need to free objects explicitly.
- This eliminates a common cause of errors in C/C++ and other languages (memory leaks). Never have to worry about *dangling references*.
- When an object is no longer reachable the space it occupies can be reclaimed.
- Space is reclaimed at the garbage collector's discretion.
- Creating and collecting large numbers of objects can interfere with time-critical applications.

finalize

- A class can implement a *finalize* method.
- This method will be executed before an object's space is reclaimed.
- Gives you a chance to use the state of the object to reclaim other non-Java resources.
- finalize is declared like this:

```
protected void finalize() throws Throwable {
   // ...
}
```

- Important when dealing with non-Java resources, such as open files.
- Example: a class that *opens* a file should provide a `close()` method. Even then, there is no guarantee that the programmer will call the `close()` method, so it should be done in a finalize method.

```
public void close()
{
   if (file != null)
   {
      file.close();
```

```
            file = null;
        }
    }
    protected void finalize() throws Throwable
    {
        try
        {
            close();
        }
        finally
        {
            super.finalize();
        }
    }
```

- The close method is written carefully in case it is called more than once.
- `super.finalize` is called to make sure your superclass is also finalized.
- Train yourself to always do this.

2.8.15 The `toString()` Method

- If an object supports a public toString method that takes no parameters and returns a String object, that method is invoked whenever a `+` or `+=` expression has an object of that type where a String object is expected.
- All primitive types are implicitly converted to String objects when used in String expressions.

2.8.16 Native Methods

- Used when you need to manipulate some hardware directly, or execute code not written in Java.
- Portability and safety of the code are lost.
- Implemented using an API provided by the people who wrote the virtual machine on the platform where the code will run.

- The standard API for C programmers is called Java Native Interface (JNI).
- Other API's are being defined for other languages.

2.9 Methods and parameters

2.9.1 Methods

- Methods in Java define the behavior of the class.
- Methods are similar to procedures or subroutines in other languages.
- The real benefits of object orientation come from hiding the implementation of a class behind its operations.
- Methods access the internal implementation details of a class that are hidden from other objects.
- Hiding data behind methods is so fundamental to object orientation it has a name – *encapsulation*.
- Methods have zero or more parameters.
- A method can have a return value.
- A method's statements appear in a block of curly braces { and } that follow the method's signature.

```
public void speak ()
{
    System.out.println("Hey Barney ...");
}
```

Invoking a Method

- Provide an object reference and the method name, separated by a dot (.):

```
fred.speak();
```

- Parameters are passed to methods as a comma-separated list.
- A method can return a single value as a result, such as an `int`:

```
public int add (int a, int b)
{
    return a+b;
}
```

- A method can return no result:

```
public void setFirstName (String firstName)
{
    this.firstName = firstName;
}
```

The this Reference

- Occasionally the receiving object needs to know its own reference.

```
public void move (double x, double y)
{
    this.x = x;
    this.y = y;
}
```

2.10 Extending Classes

2.10.1 Introduction

- It is easy for one Java class to extend the behavior of another Java class.
- This capability is usually referred to as *polymorphism*, which means that an object of a given class can have multiple forms.

2.10.2 Objectives

- Upon completion of this section, students will be able to:
- Extend existing classes.
- Define the terms subclass, superclass, overloading, and overriding.
- Define how constructors of subclasses operate.
- Define the implications of marking a class or method as `final`.
- Discuss the considerations of cloning classes.

2.10.3 An extended class

- An extended class can be used wherever the original class was legal.
- Polymorphism – an object of a given class can have multiple forms.
- The extended class is a subclass.
- The class it extends is the superclass.
- If a class does not explicitly extend a class then it implicitly extends Object.
- Object declares methods that are implemented by all objects.

2.10.4 A simple example

- A simple example of extending a base `Animal` class:

```
public abstract class Animal
{
  protected boolean hasColdNose;

  public abstract String speak();

  public void setHasWetNose(boolean hasColdNose) {
    this.hasColdNose = hasColdNose;
  }
  public boolean hasColdNose() {
    return hasColdNose;
  }
}

public class Dog extends Animal
{
  public Dog() {
    hasColdNose = true;
  }
  public String speak() {
    return "woof";
  }
}

public class Bird extends Animal
{
  public Bird() {
    hasColdNose = false;
  }
  public String speak() {
    return "chirp";
  }
}
```

```
public class Main
{
  public Main() {
    Animal fido = new Dog();
    Animal bigbird = new Bird();
    System.out.println( "fido says " + fido.speak() );
    System.out.println( "bigbird says " + bigbird.speak() );
  }

  public static void main(String[] args) {
    new Main();
  }
}
```

2.10.5 What protected really means

- A protected class member can be accessed by classes that extend that class.
- A protected class member also be accessed from code within the same package.
- A protected class member also be accessed from references of the class's type or one of it's subtypes.

2.10.6 Constructors in extended classes

- When you extend a class, the new class must choose one of it's superclass's constructors to invoke.
- If you do not invoke a superclass constructor as your constructors first executable statement, the superclass no-arg constructor is automatically invoked before any statements in your new constructor are executed.
- If the superclass does not have a no-arg constructor, you must explicitly invoke one of the superclass's other constructors, or invoke another of your own constructors using the this construct.
- If you use super(), it must be the first executable statement of the constructor.
- The language provides a default no-arg constructor for you.

Constructor order dependencies

- When an object is created, all it's fields are set to default initial values.
- Then the constructor is invoked.

Constructor phases

- Invoke a superclass's constructor.
- Initialize the fields using their initializers and any initialization blocks.
- Execute the body of the constructor.

Constructor phase example

- An example of how parent/child constructors work:

```
public class ParentClass {
  public ParentClass() {
    System.out.println( "ParentClass constructor was called" );
  }
}
public class ChildClass extends ParentClass {
  public ChildClass() {
    System.out.println( "ChildClass constructor was called" );
  }
}
public class Main {
  public static void main(String[] args) {
    ChildClass cc = new ChildClass();
  }
}
```

2.10.7 Overriding methods, hiding fields, and nested classes

Overloading – providing more than one method with the same name but with different signatures. *Overriding* – replacing a superclass's implementation with your own.

Overriding

- Signatures must be identical.
- Return type must be the same.
- Only accessible non-static methods can be overridden.
- A subclass can determine whether a parameter in an overridden method is final.

The `super` keyword

- Available in all non-static methods of a class.

`super.method()` – uses the superclass's implementation of `method`.

2.10.8 Marking methods and classes `final`

- No extended class can override the method to change its behavior. (This is the final version of that method.)
- A class marked final cannot be extended by any other class (and, all the methods of a final class are implicitly final).
- Security – anyone who uses the class can be sure the behavior will not change (validatePassword example).
- Serious restriction on the use of the class.
- Final simplifies optimizations.

2.10.9 The `object` class

- All classes inherit from Object.
- Object's methods fall in two classes: utilities, and thread support.

Object's utility methods

- `equals(Object o)`
- `hashCode()`
- `clone()`

- `getClass()`
- `finalize()`

2.10.10 Anonymous classes

- Use these when the weight of a full class seems too much for your needs.
- Extend a class or implement an interface.
- Defined at the same time they are created with new.
- Defined in the new expression.
- Cannot have their own constructors.
- Can easily become hard to read.
- Avoid anonymous classes longer than six lines.
- Use them only in the simplest of expressions.

2.10.11 Abstract Classes and methods

- Helpful when some of the behavior is defined for most objects of a given type, but some behavior makes sense for only particular classes and not the superclass.
- Declare classes that define only part of an implementation.
- Each method not implemented in an abstract class is marked abstract.
- A class with any abstract methods must be declared abstract.
- Abstract methods must be implemented by subclasses that are not abstract themselves.

2.10.12 Cloning objects

- A clone method returns a new object whose initial state is a copy of the current state of the object on which clone was invoked.
- Subsequent changes to the clone will not affect the state of the original.
- Object.clone()

Three major considerations in writing a clone method

- Empty Cloneable interface.
- Object.clone() method.
- CloneNotSupportedException.

Four different attitudes a class can have towards clone

- Support clone.
- Conditionally support clone.
- Allow subclasses to support clone but don't publicly support it.
- Forbid clone.

2.10.13 Extending classes: how and when

- Ability to extend classes – large part of the benefits of OOP.
- Creates an "IsA" relationship, not a "HasA" relationship.
- Employee / Role example (see text).

2.10.14 Designing a class to be extended

- Choose the access for each part of a class design carefully: public, protected, private.
- If your design will have subclasses, design your protected parts carefully.

Bad effects of public fields

- Fields can be modified at any time by a programmer.
- No way to add functionality.

Non-final classes have two interfaces

- Public interface for programmers *using* the class.
- Protected interface for programmers *extending* the class.

2.11 Interfaces

2.11.1 Introduction

Interfaces are a way to declare a type consisting only of abstract methods and related constants, classes, and interfaces. An interface in Java is an expression of pure design, whereas a class is a mix of design and implementation.

2.11.2 Objectives

Upon completion of this section, students will:

- Describe an interface.
- Define the differences between single and multiple inheritance.
- Show how to implement an interface.
- Describe the differences between interfaces and abstract classes.

2.11.3 An example interface

- Java has single inheritance of implementation – you can extend only one class.
- Java has multiple interface inheritance.
- All methods in an interface are implicitly abstract.
- Each class that implements the interface must implement all its methods.
- Methods in an interface are always public.
- Fields in an interface are always static and final.

Nested classes and interfaces

- Nested classes and interfaces let you associate types that are strongly related to an interface inside that interface.
- Any class or interface inside an interface is public.
- Any classes nested inside an interface are also static.

- Any interface nested inside a class can be public, protected, package-accessible, or private.

2.11.4 Single inheritance versus multiple inheritance

- A new class can extend exactly one superclass.
- The new class inherits the superclass's (a) contract and (b) implementation.
- Some languages allow multiple inheritance – two or more superclasses.
- Problem of multiple inheritance arises from multiple inheritance of implementation. (Usually related to a class that maintains state information.)

2.11.5 Extending Interfaces

- Interfaces can be extended using the extends keyword.
- Interfaces can extend more than one interface:

```
interface Shimmer extends FloorWax, DessertTopping \{ \ldots
```

- All methods and constants defined by FloorWax and DessertTopping are part of Shimmer.

Name Conflicts

- A class or interface can be a subtype of more than one interface.
- What happens when a method of the same name appears in more than one interface?

2.11.6 Implementing Interfaces

- Most interfaces may have several useful implementations.
- Enumeration is an interface.

2.11.7 Using an Implementation

- Use an implementing class just by extending it.
- Can also use a technique called forwarding – create an object of an implementing class and forward all the methods of the interface to that object.

2.11.8 Marker Interfaces

- Do not declare any methods – mark a class as having some general property.
- Define no language-level behavior.
- All of the contract is in the documentation.

2.11.9 When to Use Interfaces

Two Important Differences Between Interfaces and Abstract Classes

- Interfaces provide a form of multiple inheritance. A class can extend only one other class.
- Interfaces are limited to public methods and constants with no implementation. Abstract classes can have a partial implementation, protected parts, static methods, etc.

Interface or Abstract Class

- These two differences usually direct the choice.
- If multiple inheritance is important or even useful interfaces are used.
- Abstract class lets you define some or all of the implementation.
- Any major class you expect to be extended should be an implementation of an interface.

2.12 Exceptions

2.12.1 Introduction

- Applications can run into many kinds of errors during execution.
- Java exceptions provide a clean way to check for errors without cluttering code, and provide a mechanism to signal errors directly.
- Exceptions are also part of a method's contract.
- An exception is thrown when an unexpected error condition is encountered.
- The exception is then caught be an encompassing clause further up the method invocation stack.

2.12.2 Objectives

Upon completion of this section, students will be able to:

- Create their own Exception class.
- Throw an exception.
- Define the three choices you have when a method throws an exception.
- Use try/catch/finally to run a method that may throw an exception.
- Describe the purpose of the finally clause, and when it is run.
- Make better decisions about when to throw exceptions in your code.

2.12.3 Creating exception types

- Exceptions are objects.
- Must extend the class `Throwable` or one of it's subclasses.
- By convention new exception types extend `Exception` (a subclass of `Throwable`).

```
public class BadPizzaException extends Exception
{
   public String attrName;
```

```
    BadPizzaException(String name)
    {
       super( "BadPizzaException: " + name );
       attrName = name;
    }
}
```

2.12.4 throw

- Exceptions are thrown using the `throw` statement.
- Exceptions are objects, so they must be created (with `new`) before being thrown.

2.12.5 The throws clause

- The exceptions a method can throw are declared with a throws clause.
- The exceptions a method can throw are as important as the value type the method returns.

Choices when invoking a method that has a throws clause

- Catch the exception and handle it.
- Catch the exception and map it to one of your exceptions by throwing an exception of a type declared in your own throws clause.
- Declare the exception in your throws clause and let the exception pass through your method.

2.12.6 try, catch, and finally

- Exceptions are caught by enclosing code in try blocks.
- The body of try is executed until an exception is thrown or it finishes successfully.
- If an exception is thrown, each catch clause is examined in turn, from first to last, to see whether the exception object is assignable tto the type declared with the catch.
- When an assignable catch is found, its code block is executed. No other catch clause will be execu

- Any number of catch clauses can be associated with a try, as long as each clause catches a different type of exception.
- If a finally clause is present in the try block, the code is executed after all other processing in the try is complete. This happens no matterhow the try clause completed – normally, through an exception, or through a return or break.

finally

- A mechanism for executing a section of code whether or not an exception is thrown.
- Usually used to clean up internal state or to release non-object resources, such as open files stored in local variables.
- Can also be used to cleanup for break, continue, and return.
- No way to leave a try block without executing its finally clause.
- A finally clause is always entered with a reason; that reason is remembered when the finally clause exits.

2.12.7 When to use exceptions

- Used for "unexpected error conditions".
- Not meant for simple expected situations (end of an input stream should be expected).

2.13 Packages

2.13.1 Introduction

- Packages have members that are *related* classes, interfaces, and subpackages.
- Packages are useful for several reasons:
- Packages create a grouping for related interfaces and classes.
- Interfaces and classes in a package can use popular public names that make sense in one context but might conflict with the same name in another package.
- Packages can have types and members that are available only within the package.

2.13.2 Package Naming

- A package name should be unique.A good way to ensure unique package names is to use an Internet domain name (reversed):

```
package COM.missiondata.utils;
```

2.13.3 Package Access

- A public class or interface is accessible to code outside that package.
- Types that are not public have package scope; they are available to all other code in the same package, but are hidden outside the package and even from code in nested packages.
- Package scope is the default if public/protected/private are not declared.

2.13.4 Package Contents

- Contain functionally-related classes and interfaces.
- Classes in a package can freely access each other's non-private members.

- Should provide logical groupings for programmers looking for useful interfaces and classes.
- Packages can be nested inside other packages.

2.13.5 Examples

- The Model/View/Controller paradigm can help illustrate a good way of packaging MVC-related classes for a fictitional Pizza Store project:

```
com.alspizzastore.model.Pizza
com.alspizzastore.model.Topping

com.alspizzastore.view.OrderEntryScreen
com.alspizzastore.view.ReceiptView

com.alspizzastore.controller.ReceiptController
com.alspizzastore.controller.PricingController
```

Chapter 3

Day 3: Standard Libraries & Server-side Programming

3.1 Objectives

- Learn how to read-from and write-to files with IO facilities.
- Gain an introduction to writing network-aware programs with Java.
- Obtain a basic understanding of how to create and use threads in Java programs.
- Learn how Remote Method Invocation (RMI) works and when/where it is used.
- Learn how to use the classes in the Collections API.
- Gain an overview of creating international applications.
- Learn the basics behind the HTTP protocol.
- Learn how to use JavaServer Pages (JSPs) and Java Servlets.

3.2 IO: Streams and readers

- The java.io package provides a rich framework and class library for reading and writing any type of data (text, numbers, images) in streams from any type of location (disk, network, memory).
- The package provides 2 class hierarchies: one for byte streams, the other for character streams.
- Base classes for reading data: InputStream for byte streams, Reader for character streams.
- Base classes for writing data: OutputStream for byte streams, Writer for character streams.
- Example of copying one file to another.

```
Reader in = new FileReader("one.txt");
Writer out = new FileWriter("two.txt");
int token = -1;

while( (token != in.read()) != -1 )
{
  out.write(token);
}
out.close();
```

- The library uses the decorator pattern to add flexible stream handling. By chaining constructors of various classes, you can change how streams are read or written. Examples:
  1. We need to read some data on disk, and for performance reasons the read operation should be buffered.

     ```
     Reader bufferedFileIn =
         new BufferedReader(new FileReader("datafile"));
     ```

  2. Read an in-memory byte array as UTF-8 encoded characters.

```
byte[] data = ...;
Reader charReader =
    new BufferedReader(
      new InputStreamReader(
        new ByteArrayInputStream(data), "UTF-8"));
```

3.3 Java networking

3.3.1 Introduction

- Socket
- ServerSocket
- URL
- URLConnection

3.3.2 Socket

- `Socket socket = new Socket(host, port);`
- `server` is a `String` or `InetAddress`.
- `UnknownHostException` – could not convert the given host/server name to a TCP/IP address.
- `IOException` – could not find server or port.
- Methods: `getInetAddress()`, `getPort()`, `getLocalPort()`, `getLocalAddress()`, `getInputStream()`, `getOutputStream()`.

3.3.3 ServerSocket

- Listens for clients.
- Use `accept()` to accept incoming connections.
  - `accept` returns a Socket.
  - `Socket socket = serverSocket.accept();`

3.3.4 ServerSocket lifecycle

- A new `ServerSocket` is created on a port using a `ServerSocket()` constructor.
- `ServerSocket` listens on the port using the `accept()` method.
- Uses `getInputStream()` and/or `getOutputStream()`.
- Server and client interact using an agreed-upon communication protocol.

- Either the server or the client (or both) close the connection.
- The server goes back to listening with the `accept()` method.

3.3.5 URL

- URL page = new URL("http://www.devdaily.com");
- `getContent()`
- `openStream()`
- `openConnection()` – returns a `URLConnection`.

3.3.6 URLConnection

- Created from a `URL` object using `openConnection()`.

```
URL url = new URL("http://www.devdaily.com");
URLConnection uc = url.openConnection();
```

- `getInputStream()` – read data from the server.
- `getContentType()` – returns the MIME content type of the data.
- `getContentLength()` – returns the number of bytes in the content.

3.4 Threads

3.4.1 Objectives

- Explain what threads are, how they work, and when to use them.
- Create threads using the Thread class.
- Create threads using the Runnable interface.
- Describe thread states.
- Understand the methods that are used when manipulating threads.

3.4.2 Applications without multiple threads

- Many applications work without multiple threads of concurrency.
- In this world, a sequence of actions is completed in a linear, one-thread-of-thought-at-a-time manner.
- Java makes it easy for developers to create multiple threads that essentially run simultaneously. (How this *really* works depends on what operating system your application is running on.)
- A thread is considered *lightweight* because it runs within the context of a full-blown program.

3.4.3 Thread states

- new – an empty thread, with no system resources allocated; all you can do is start it.
- runnable – the `start()` method is called, and the thread is not dead or in the "not runnable" state.
- not runnable – `sleep()` is invoked, thread calls `wait()`, thread is blocking on I/O.
- dead – the end of the `run()` method has been reached.

3.4.4 Creating a threaded class with `thread`

- Extend the `java.lang.Thread` class and override its `run()` method.
- The `run()` method defines the behavior of the thread while it is running.
- When the `run()` method ends, the thread goes into the dead state.
- As a developer, you call the `start()` method of the thread; the JVM calls the `run()` method.
- Never call the `run()` method directly.

3.4.5 Creating a threaded class with the `runnable` interface

- Cannot always extend the Thread class because of single inheritance.
- Implement the `java.lang.Runnable` interface (must implement the `run()` method).
- Include a `Thread` attribute in the class.
- The `run()` method still defines the behavior of the thread while it is running.

3.4.6 Thread methods

- `sleep(long ms)` – causes the thread to sleep for the specified number of milliseconds.
- `yield()` – provides a hint to the scheduler that this doesn't have to run at the current time, so the scheduler can choose another thread to run if need be.
- `join()` – used to let one thread wait for another to terminate.
- `interrupt()` – used to request that the thread cancel itself.
- `notifyAll()` – wakes up all waiting threads.
- `wait()` – used to pause the thread, presumably while it waits for some condition to be satisfied.

3.4.7 Thread references

- The Java Programming Language – excellent chapter on threads.
- Sun's Java Tutorial, at http://java.sun.com/docs/books/tutorial/, or more specifically http://java.sun.com/docs/books/tutorial/essential/threads/index.html.

3.5 JavaBeans

- A JavaBean is a reusable software component based on Sun's JavaBeans specification that can be visually manipulated in a builder tool.
- A JavaBean is simply a Java class that has get and set methods for each class attribute.
- These get and set methods follow a standard naming convention.
- A set of properties that can be read, changed, or discovered.
- May be serializable.
- May generate or listen to events.

3.6 Remote Method Invocation (RMI)

- Allows Java programs to call methods on Java objects in different virtual machines, and even different physical machines.
- An RMI server is a java class that is registered in the JVM.
- The JVM listens on a dedicated port for RMI requests.
- RMI clients call the server through a proxy that implements the same interface as the server.
- The Java SDK contains the rmic compiler to automatically generate the proxy and server code required to make a class an RMI server.

3.7 Java Native Interface (JNI)

- Lets you run C programs from Java.
- A great feature if you have a large amount of high-quality C code already written, and want to leverage that for new Java applications.
- Can also be used if maximizing the speed of the code is critical.
- When using JNI, you may be limiting the portability of your application.

3.8 Collections framework

- The java.util package provides interfaces and implementations for standard data structures:
  - List
  - Map
  - Set
- Collection is the base interface for all collection types, with methods for adding, removing, and counting elements.
- Iterator - Interface for iterating over a collection.
- The collections have a single element type: Object. This has two side effects:
  - Primitives such as int, char, and boolean cannot be placed in collections without using their object wrappers (Integer, Character, Boolean).
  - It is up to the programmer to keep collections consistent, since one can add objects of multple types to a collection.
- Unless explicitly specified in the documentation, assume Collection implementations are NOT thread safe. Why use the unsafe implementations at all? Because unless you really are sharing the collections among threads, they will be considerably faster than the thread-safe versions.
- Basic collection usage.

```
Collection stuff = new HashSet();
String kiwi = "Kiwi";

stuff.add("Apple");
stuff.add("Orange");
stuff.add(kiwi);

Iterator it = stuff.iterator();

while(it.hasNext())
```

```
{
    String item = (String)it.next();
    System.out.println(item);
}
```

3.8.1 Lists

- In addition to the standard collection methods, the List interface allows for random access of elements through its get and set methods.
- List indices are zero based, so list.get(list.size())) is not a valid element.

```
List stuff = new ArrayList();

stuff.add("Apple");
stuff.add("Orange");

// what's the last item? Indexes are zero based!
System.out.println(stuff.get(stuff.size() - 1));

// change the first item to a pear.
stuff.set(0, "Pear");

// now iterate
Iterator it = stuff.iterator();

while(it.hasNext())
{
   String item = (String)it.next();
   System.out.println(item);
}
```

- ListIterator allows element addition, removal, and replacement during iteration.
- The java.util package provides two new List implementations: ArrayList and LinkedList.
- Vector is a legacy class retrofitted to the new Collection framework, and is thread safe.

3.8.2 Maps

- Maps associate the data elements in the collection with keys. The keys are derived from type Object, and may be different a type than the data. The get and put methods provide access to elements at a particular key.

```
Map userAges = new HashMap();

ages.put("John", new Integer(25));
ages.put("Nicole", new Integer(32));
ages.put("Bob", new Integer(43));

// How old is Bob?
Integer bobsAge = (Integer)userAges.get("Bob");

// if we have a user named Nicole, what's her age?
if(userAges.containsKey("Nicole"))
{
 Integer nicolesAge = (Integer)userAges.get("Nicole");
}
```

- Caution: If the key object is mutable, take care not to change its value after using it in a Map.
- There are several Map implementations including HashMap and Hashtable.
- As a Java 1.1 legacy class, Hashtable is syncronized for thread safety.

3.8.3 Collection Utilities

- The java.util.Collections class provides standard utilities for Collections.
  - List Sorting
  - Binary List Search
  - List reversing and shuffling
  - Min and Max
  - Synchonization

- Sorting makes use of the Comparator interface, which a programmer implements to determine how objects of a particular type are to be ordered.
- The Syncronization methods return thread-safe collections that are backed by a specified collection. Note that iteration is still not thread safe in this case.

3.9 Internationalization, localization, and formatting

- `Locale` – a specific place, cultural, political, or geographical.
- Objects can localize their behvior to a user's expectations (*locale-sensitive* object).
- `ResourceBundle` – methods to look up resources in a bundle by a string key.

3.10 HTTP protocol

- Underlying protocol for all Web Browser/Web Server communication whether static or dynamic.
- Uses request and response communication model.
- Cookies

3.10.1 Request and Response

- Client issues a request to the server.
- Server processes the request, and responds to the client.
- The GET verb asks the server for the content at a given URL. The client may specify extra parameters for the server. These query parameters can be seen on the URL in the browser: "http://www.yahoo.com/search?q=java".
- The POST verb sends data to the server to be processed. This data is packaged in the request a well defined format, and is not visible in the browser. The server responds in the same way as with GET requests.
- Headers are name value pairs that precede the request or response data. These can indicate such informations as what type of data is being sent, how long it is, and what type of client/server is doing the sender.
- Example:
  - At the command prompt: telnet www.google.com 80
  - When connected type exactly: GET /index.html HTTP/1.1
  - Hit enter twice - this sends the request.
  - Note the response code - "200 OK" means the server processed the request successfully.
  - Note the headers: Content-Type, Content-Length, a cookie, and the web server type.

3.10.2 Cookies

- Simple name/value pairs used to store state on the client.
- Cookies are set when the server sends a Set-Cookie header in a response.
- The browser sends the server the cookies back in the request headers.
- The server is only sent the cookies for its domain.
- Many web application platforms (JSP, servlets, ASP) use cookies for session tracking.
- Cookies have expiration dates after which they are no longer sent to the server.
- The total cookie header may not exceed 4K, and client sofware may have more restrictions.
- Due to security concerns some users turn off cookies in their browsers.

3.11 Servlets and JSPs

3.11.1 Objectives

- Describe the JSP dynamic HTML model.
- Write JSPs to generate dynamic HTML.
- Include Java code in JSPs.
- Understand and use implicit objects in JSPs.

3.11.2 Introduction/Background

- Original HTML content on the web was largely static HTML pages.
- CGI scripts were the first way of generating HTML pages with dynamic content.
- Servlets are Java programs that run within a web server, or within a web application server.
  - There is a special Servlet API.
  - Servlets work hand-in-hand with databases, JavaBeans, components, and JSPs.
- JSPs are a combination of HTML and Java code.
- JSPs are a form of servlets; that is, they are compiled into servlets behind the scenes. In practice however, they appear quite different, and appear like a combination of HTML and Java code.
- In practice, JSPs should be approximately 90% HTML code and 10% Java code.

3.12 Servlets

3.12.1 Objectives

- Understand servlet framework.
- Write basic servlets.

3.12.2 Servlet basics

- Used to deliver dynamic content to web pages.
- Request
- Response
- GET and POST
- The servlet API
- Request, service(), doGet()/doPost(), response.

3.12.3 HelloWorldServlet

- A sample HelloWorldServlet:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class HelloWorldServlet extends HttpServlet
{
  public void doGet(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, IOException
  {
    resp.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<html><body>Hello, world</body></html>");
    out.close();
  }
}
```

3.12.4 Servlet lifecycle

- Handled by the servlet container.
- Create and initialize the servlet.
- Handle zero or more service calls.
- Destroy and garbage collect the servlet.
- A single servlet instance to handle every request.

3.12.5 `HTTPServlet`

- Override `doGet()` to handle GET requests.
- Override `doPost()` to handle POST requests.
- Both methods take `HTTPServletRequest` and `HTTPServletResponse` as arguments.
- A few other methods, but used much less often – `doDelete`, `doTrace`, `doOptions`, `doPut`

3.12.6 `HTTPServletRequest`

- `getMethod()`
- `getQueryString()`
- `getRemoteHost()`
- `getRemoteAddr()`
- `getAuthType()`
- `getContentType()`

3.12.7 `HTTPServletResponse`

- `getWriter()`
- `setContentType()`
- `getOutputStream()`

3.13 JavaServer Pages

- What is a JSP?
- Why use JSPs?
- Directives
- Scriptlets
- Actions

3.13.1 What is a JSP?

- JSPs are a combination of HTML code and Java code.
- JSP files usually have a filename extension of ".jsp".
- JSP files are compiled into Servlets by the JSP container.
- Provides automatic session management.
- Implicit programming objects are available, including `request`, `response`, and others.
- JavaBeans are the component model.

3.13.2 JSP engine/container:

- Compiles a JSP page into a servlet (once, the first time it is called).
- Eexecutes the servlets `service()` method.
- Sends the resulting text back to the caller.

3.13.3 Translation time and request time

- Translation time – the time a JSP is compiled into a Servlet.
- Translation time – certain JSP elements are evaluated at translation time.
- Request time – the time a JSP is requested by a user.
- Request time – some JSP elements, such as expressions, are evaluated at request time.

3.13.4 Scriptlets

- Use `<% ... %>` syntax to include Java code directly into the JSP.
- Any valid Java code can be included.

3.13.5 Expressions

- Use `<%= ... %>` syntax to include expressions into a JSP.
- An expression is a piece of Java code that evaluates to some thing or some value.
- Example: `<%= 2+2 %>` evaluates to 4.
- Expressions are evaluated at request time.

3.13.6 Declarations

- Use `<%! ... %>` syntax for declarations.
- Use declarations to declare class scope variables and methods.

3.13.7 Directives

- Use `<%@ ... %>` syntax for declarations.
- Use directives to declare information needed by the JSP engine.
- Directives include `page`, `include`, and `taglib`.

`page` directive

- `<%@ page contentType="text/html" %>`
- Lets the developer specify packages to include.
- Lets the developer specify more advanced page features.

request	Represents the HTTP request as received by the server.
response	Represents the HTTP response to be send by the server.
out	PrintWriter object that is used to write to the response.
err	poop

Figure 3.1: List of implicit JSP objects

`taglib` directive

- `<%@ taglib uri="uriToTaglib" prefix="someID" %>`
- Lets the developer include tag libraries.
- Tag libraries are generally powerful, reusable components for web developers.
- Tag libraries can be created by local developers, are available open source, or can be purchased.

`include` directive

- `<%@ include file="filename.jsp" %>`
- Lets one JSP include another JSP or HTML document.
- The JSP container reads the included file, and creates on servlet.
- Includes occur at translation time.
- If you change the included file ...
- Works differently than `<% jsp:include page="filename.jsp" />`

3.13.8 Implicit objects

- As a JSP developer, certain implicit objects are made available to you.
- The JSP contaner provides them for you.
- Some objects, such as request and response, are passed as parameters to the servlets service() method.

The table below provides details about each implicit object.

3.13.9 Exception handling

- If a JSP causes an exception to be thrown, good practices dictate that the exception is handled by code, or that the user be forwarded to an error page.
- Exceptions can be caught in scriptlets.
- Use the `<%@ page errorPage="/error.jsp" %>` directive to define an error page.
- Identify the error page with the page directive, `<%@ page isErrorPage="true" %>`.
- The JSP container makes the implicit `exception` object available to a defined error page.

3.14 Survey of other server-side Java technologies

- XML
- XSLT
- Enterprise Java Beans
- Java Messaging Service

3.14.1 XML

- eXtensible Markup Language
- Derived from SGML, which is also the original parent language of HTML.
- XML is a "meta" language, that can be used to defined an unlimited number of custom hierarchical data formats.
- Example:

```
<?xml version="1.0" ?>
<sandwich price="2.00">
  <meat>Pastrami</meat>
  <bread>Rye</bread>
  <cheese slices="1">Swiss</cheese>
  <condiments>
    <condiment>mustard</condiment>
    <condiment>pepper</condiment>
  </condiments>
</sandwich>
```

- In the example above `<sandwich>`, `<condiment>`, and all tags within `<>` are elements.
- Elements can be nested within each other infinitely deep.
- "price" and "slices" are attributes of their respective elements.
- XML must be well formed to be porcessed:
  - All tags are closed: `<bread>Rye</bread>`,
    or `<bread/>` which denotes an empty element.

  - Elements must be closed on the same level they start. `<sandwich><meat></sandwich></meat>` is not well formed.
  - All attributes are within single or double quotes.

- The structure of a particular XML document can be described with Document Type Definitions (DTD) or schemas. Instances of that document can then be validated against the corresponding DTD or schema.

- Why use XML?

  - Flexible hierarchical data formatting language that can easily be read by humans as well as programs.
  - Readily available tools for document creation, parsing, and validation.
  - Plaform and programming language neutral.

- The Java API for XML Processing (JAXP) defines a standard interface for using XML in Java programs. There are several implementations of this API, including Xerces from the Apache Group and Crimson from Sun.

3.14.2 XSLT

- eXtensible Stylesheet Language: Transformation

- XSLT documents are instructions that describe how to transform one XML document into another type of XML document.

- Example uses:

  - Converting the XML for an order into HTML that displays the order to the user.
  - Converting an XML invoice from a vendor's system into XML that your internal system understands.

- This sample shows how to transform the sandwich XML from the previous section into HTML.

```
Input XML:
<?xml version="1.0" encoding="UTF-8"?>
<sandwich price="2.00">
```

```
  <meat>Pastrami</meat>
  <bread>Rye</bread>
  <cheese slices="1">Swiss</cheese>
</sandwich>

XSLT:
<!-- XSLT is itself XML -->
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/sandwich">
  <!-- the HTML tags are embedded directly in the template -->
  <html>
  <head><title>My Sandwich</title></head>
    <body>
    <!-- when executed, the value-of element is replaced
         with the actual value -->
    <b>Meat:</b> <xsl:value-of select="meat" /><br />
    <b>Bread:</b> <xsl:value-of select="bread" /><br />
    <b>Cheese:</b> <xsl:value-of select="cheese/@slices" />
        slice(s) of <xsl:value-of select="cheese" />
    <br /><br />
    <b>Cost:<xsl:value-of select="@price" /></b>
    </body>
  </html>
</xsl:template>
</xsl:stylesheet>
Output:
<html>
<head>
  <title>My Sandwich</title>
</head>
<body>
  <b>Meat:</b>Pastrami<br />
  <b>Bread:</b>Rye<br />
  <b>Cheese:</b>1 slice(s) of Swiss
  <br /><br />
  <b>Cost: 2.00</b>
 </body>
</html>
```

- The JAXP defines the API for executing XSLT transformations.

3.14.3 Enterprise Java Beans

- Allows the developer to write components without explicitly coding for database storage, transactions, thread safety, or remote invocation.
- XML configuration files are used to mark how components are mapped to databases, which methods require transactions, and how the components can be accessed.
- The EJB specification is implemented by many vendors including IBM, Borland, BEA, Oracle, and Macromedia. There are also a few open source implementations.

3.14.4 Java Messaging Service

- Provides asynchronous message based communication to Java applications.
- There are two styles of JMS:
  - Queues - Messages are stored in queues where they can be processed in the order received.
  - Publish and Subscribe - Multiple subscribers are notified when messages are published.
- JMS can guarantee message delivery
- Messages can be processed as part of a transaction. If the transaction is rolled back, the message activity is also rolled back.

Chapter 4

Day 4: Databases, Best Practices, and Final Project

Getting Started Setting Up a Database Establishing a Connection Setting Up Tables Retrieving Values from Result Sets Updating Tables Milestone: The Basics of JDBC Using Prepared Statements Using Joins Using Transactions Stored Procedures SQL Statements for Creating a Stored Procedure

4.1 Databases and JDBC

4.1.1 Getting things set up

What you need to get started:

- Java JDK (and JDBC).
- JDBC driver for your database.
- A database server (Oracle, Informix, DB2, Postgres, etc.).
- Create the database and tables you want to use.

4.1.2 Connecting to the database

Load the driver

- `Class.forName("jdbc.DriverXYZ");`
- `Class.forName("org.postgresql.Driver");`

Create the connection

- `Connection conn = DriverManager.getConnection(url,"loginID", "password");`
- The URL:
  - `jdbc:postgresql:database`
  - `jdbc:postgresql://host/database`
  - `jdbc:postgresql://host:port/database`

4.1.3 Statements

- A `Statement` object is used to send an SQL statement to the DBMS.
- Create a `Statement` object and then execute it, using the proper `execute` method:
  - For SELECT statements use `executeQuery`.
  - For statements that create or modify tables use `executeUpdate`.
- Example:

```
Statement stmt = conn.createStatement();
String query = "SELECT username, password FROM user";
ResultSet rs = stmt.executeQuery(query);
while ( rs.next() )
{
  String user = rs.getString("username");
  String password = rs.getString("password");
  System.out.println( "Username: " + user );
  System.out.println( "Password: " + password );
}
```

4.1.4 getXXX methods

With `ResultSet` objects:

- `getByte`
- `getShort`
- `getInt`
- `getLong`
- `getFloat`
- `getDouble`
- `getBigDecimal`
- `getBoolean`
- `getString`
- `getBytes`
- `getDate`
- `getTime`
- `getTimestamp`
- `getAsciiStream`
- `getUnicodeStream`
- `getBinaryStream`
- `getObject`

4.1.5 Updating the database

Use `executeUpdate` when using SQL UPDATE commands:

```
String update = "UPDATE user SET password='bar' WHERE user='foo'";
stmt.executeUpdate(update);
```

4.1.6 `PreparedStatements`

- Developers often use `PreparedStatements` because the syntax is easier.
- Example:

```
PreparedStatement update =
   conn.prepareStatement("UPDATE user SET password = ? WHERE user = ?");
updateSales.setString(1, "bar");
updateSales.setString(2, "foo");
```

- Intended for, most useful, and high performance, inside of loops where many INSERT's or UPDATE's need to be done at one time.

4.1.7 A real method

The following method was copied from a production software application. Note that it uses a `PreparedStatement` for the syntactical ease of use that the `PreparedStatement` offers.

```
public String getPurchaserEmailAddress(int orderId)
throws SQLException
{
  Connection connection = null;
  try
  {
    String email = null;
    connection = ConnectionPool.getConnection();
    String query = "SELECT email FROM orders WHERE order_id = ?";
    PreparedStatement emailQuery = connection.prepareStatement(query);
    emailQuery.setInt(1, orderId);
    ResultSet rs = emailQuery.executeQuery();
    if ( rs.next() )
    {
      email = rs.getString(1);
    }
    return email;
  }
  finally
  {
    ConnectionPool.freeConnection(connection);
  }
}
```

4.2 JUnit

4.2.1 Is Testing Important?

Do you think testing your software code is important? Consider the following examples:

4.2.2 Mars Orbiter

A failure to recognize and correct an error in a transfer of information between the Mars Climate Orbiter spacecraft team in Colorado and the mission navigation team in California led to the loss of the spacecraft last week, preliminary findings by NASA's Jet Propulsion Laboratory internal peer review indicate... The peer review preliminary findings indicate that one team used English units (e.g., inches, feet and pounds) while the other used metric units for a key spacecraft operation. This information was critical to the maneuvers required to place the spacecraft in the proper Mars orbit.

4.2.3 USS Yorktown

An article in the November 1998 Scientific American describes an incident aboard the USS Yorktown, a guided missile cruiser. A crew member mistakenly entered a zero for a data value, which resulted in a division by zero, an error that cascaded and eventually shut down the ship's propulsion system. The Yorktown was dead in the water for a couple of hours because a program didn't check for valid input.

4.2.4 Types of tests

We can't tell the story of working backwards (using JUnit) until we see how and why we should use tools like JUnit. So first, a little background on testing.

- **Unit tests** – An automated exercise of code where the developer may create an artificial environment to automatically exercise a unit of code.
- **Regression tests** – Running the same set of tests regularly or after known events.
- **Black Box tests** – Testing based only upon the documentation of the unit without any knowledge of the implementation.

- **White Box tests** – Tests based on knowledge of the internals of the unit.
- **Integration tests** – Taking separate units of software that have already been tested and testing them together.
- **Bounds tests** – Testing the outer ranges of values. Nulls, zeroes, blanks, etc.
- **Stress tests** – Testing a system's ability to handle a desired system load, typically more than is expected.
- **System/Acceptance testing** – Making sure the software as a whole works as expected.

4.2.5 Unit Testing 101

Definitions of unit testing

A few definitions of unit tests from various sources ...

- A unit test is an automated exercise of code.
- A unit test is written for each module (class), in isolation, to verify its behavior.
- Unit tests are tests that you, the developer, create to exercise your code and prove that it works, i.e., prove that each of your methods meet the criteria of its contract.
- Unit testing is a form of "white box" testing.
- In combination, unit tests can be used to create large regression tests.

C3 (the famous Chrysler C3 project) has over 1300 unit tests, performing over 13,000 individual checks. They run in about 10 minutes.

4.2.6 Goals of unit testing?

- Test each part (unit) of the code in isolation.
- Create tests that retain their value over time.
- Create much larger regression tests that help demonstrate that the entire system still works (or not) after changes.

- In the XP view, *if a program feature lacks an automated test it is assumed that it doesn't work.*
- In a team environment unit tests ensure that we don't break one another's code.

4.2.7 Unit Testing with JUnit

How to create unit tests with JUnit

1. Build a subclass of the class `TestCase`.
2. Create as many test methods in this test class as necessary.
3. With JUnit, if each method begins with the name "test", like `testThis()`, `testThat()`, etc., the test methods will be discovered automatically.
4. The testing framework collects up all the tests and executes them one after another.
5. You can create a method named `setUp()` that sets stuff up before each test is run.
6. You can create a method named `tearDown()` that helps you get rid of stuff after each test is run.

4.2.8 A sample JUnit session

Let's go back to the need for a `Pizza` class. Here are the things we want to be able to do with our `Pizza` class.

1. Create a means of adding toppings to a pizza.
2. Create a `Pizza` class and a test class for it.
3. Create methods to get the list of toppings from a pizza.
4. Create a way to remove a topping from a pizza.
5. Create a way to determine the price of a pizza.

For this class your first task is to make sure you can add and remove toppings. How do you start?

1. First, make sure you have JUnit set up properly so you can easily use it.

2. Next, create the desired `Pizza` class, but implement nothing.

3. Next, create a test class named `_Pizza`. This class will extend `TestCase`, and will hold all of your unit tests for the Pizza class.

4. In the _Pizza class, start to implement the first test method. The first thing I want to be able to do is to get a list of toppings that a pizza has, so I write a little code like this:

```
public void testToppingsOnNewPizza()
{
  Pizza pizza = new Pizza();
  List toppings = pizza.getToppings();
  assert( (toppings.size()==0) );
}
```

5. Run this JUnit test. Does it work?

6. No, it doesn't work, because the `getToppings()` method does not exist. So what do you do next? Make it work. Implement this behavior in the `Pizza` class.

7. Go to the `Pizza` class. Create a method named `getToppings()`. From what we know so far it should return a `List`, and apparently for a new `Pizza()`, the best thing to do is to return an empty `List` (not a `null`, but a `List` with nothing in it).

```
public List getToppings()
{
  return this.toppings;
}
```

8. Will this work by itself? No, you also have to have a `List` named toppings in the `Pizza` class. Here's what the `Pizza` class should look like:

```
public class Pizza
{
```

```
  private List toppings = new LinkedList();

  public List getToppings()
  {
    return this.toppings;
  }
}
```

9. Now run JUnit again . . . does the test work?

4.2.9 Recap

"Whoa . . . that went fast . . . how about a recap?" No problem. Here's what we did:

1. Write one test.
2. Compile the test. It should fail (because you haven't implemented anything yet).
3. Implement just enough to compile. (Refactor first if necessary.)
4. Run the test and see it fail.
5. Implement just enough to make the test pass.
6. Run the test and see it pass.
7. Refactor for clarity and "once and only once".
8. Repeat from the top.

One of the great things to come out of the recent Extreme Programming movement has been to bring unit testing to the forefront of every developers thought process.

4.3 Best practices

- Very short iterations, and immediate feedback (XP, RUP).
- Working backwards and ruthless testing (Junit).
- Use source code control (CVS).
- Use build tools (Ant).
- Communicate, communicate, communicate (XP).
- Don't live with broken windows.
- Code generators.
- On-site customer (XP).
- Don't repeat yourself (DRY).
- Business ROI.

4.4 Refactoring

Refactoring is a technique to restructure code in a disciplined way. Code that smells. `:)`

- Refactoring is a technique to restructure code in a disciplined way.
- A change to the system that leaves its behavior unchanged, but enhances some nonfunctional quality - simplicity, flexibility, understandability, performance.

Like Design Patterns, there is a long list of well-known refactorings:

- Add Parameter
- Change Reference to Value
- Change Value to Reference
- Decompose Conditional
- Duplicate Observed Data
- Encapsulate Collection
- Encapsulate Downcast
- Extract Class
- Extract Interface
- Extract Method
- Extract Subclass
- Extract Superclass
- Hide Delegate
- Hide Method
- Inline Class
- Inline Method
- Inline Temp
- Many more ...
- `http://www.refactoring.com`

4.5 Final project

Note – the final project will vary per class.

Bibliography

[1] Cockburn: Surviving Object-Oriented Projects

[2] Hunt, Thomas: The Pragmatic Programmer

[3] http://java.sun.com/docs/books/tutorial/ – Sun's online tutorials

[4] Rosenberg, Scott: Use Case Driven Object Modeling with UML, A Practical Approach

[5] Fowler: UML Distilled

[6] Schneider, Winters: Applying Use Cases

[7] Cockburn: Writing Effective Use Cases

[8] http://books.txt.com – use case templates

[9] Kruchten: The Rational Unified Process

[10] Alhir: UML in a Nutshell

[11] Visual Modeling with UML

[12] Page-Jones: Fundamentals of Object-Oriented Design in UML

[13] Arnold, Gosling, Holmes: The Java Programming Language, Third Edition

[14] Hall: Core Servlets and JavaServer Pages

[15] Sun: JDBC API Tutorial and Reference, Second Edition

[16] Fowler: Refactoring; Improving the Design of Existing Code

[17] Kernighan, Pike: The Practice of Programming

[18] http://java.sun.com

[19] http://www.xprogramming.com

[20] http://www.extremeprogramming.org

[21] http://www.junit.org

[22] http://www.martinfowler.com

[23] http://members.aol.com/acockburn – Alistair Cockburn

[24] http://www.sei.cmu.edu/cmm/cmm.html – Capability Maturity Model for Software (CMM or SW-CMM).