

BEST CODING PRACTICES

JAVA

Student Introduction

- What is your name?
- What is your experience and role/job?
- Why are you taking this course?

Objective

Day-1

- Characteristics of High-Quality Software
- What are coding best practices ?
- The pillars of OOP
- SOLID Design Principles
- Pillars of Code Quality & Code Metrics
- Collections best practices

Objective

Day-2

- Generics, Enums and Annotations
- Why Should I Test code first – Junit ?
- Introduction to PMD/CPD
- Lets help support – effective logging
- Refactoring Legacy Code
- Maximize Performance
- Effective Multithreading with thread synchronized

Software required –

- JDK 7/8
- Server – Tomcat 7/8
- IDE – Eclipse Luna
- Apache Log4J
- Junit
- PMD

Characteristics of High-Quality Software

Key characteristics of high-quality software

There are 6 main quality characteristics ([ISO 9126-1](#)) –

- **Functionality**
- **Reliability**
- **Usability**
- **Efficiency**
- **Maintainability**
- **Portability**

Key characteristics of high-quality software

■ **Functionality** -

- **Suitability** - Appropriateness (to specification) of the functions of the software.
- **Accurateness** - Correctness of the functions, an ATM may provide a cash dispensing function but is the amount correct?
- **Interoperability** - Ability of a software component to interact with other components or systems.
- **Compliance** - Appropriate certain industry (or government) laws and guidelines need to be complied with the compliant capability of software.
- **Security** - Unauthorized access to the software functions.

Key characteristics of high-quality software

■ **Reliability** -

- **Maturity** - Frequency of failure of the software.
- **Fault tolerance** - To withstand (and recover) from component or environmental, failure.
- **Recoverability** - Bring back a failed system to full operation, including data and network connections.

Key characteristics of high-quality software

■ Usability -

- **Understandability** - Ease of which the systems functions can be understood, relates to user mental models in Human Computer Interaction methods.
- **Learnability** - Learning effort for different users, i.e. novice, expert, casual etc.
- **Operability** - Easily operated by a given user in a given environment.

Key characteristics of high-quality software

- **Efficiency -**

- **Time behavior** - Response times for a given thru put, i.e. transaction rate.
- **Resource behavior** - Resources used, i.e. memory, cpu, disk and network usage.

Key characteristics of high-quality software

▪ **Maintainability** -

- **Analyzability** - Ability to identify the root cause of a failure within the software.
- **Changeability** - Amount of effort to change a system.
- **Stability** - Characterizes the sensitivity to change of a given system that is the negative impact that may be caused by system changes.
- **Testability** - Characterizes the effort needed to verify (test) a system change.

Key characteristics of high-quality software

■ Portability -

- **Adaptability** - Change to new specifications or operating environments.
- **Installability** - The effort required to install the software.
- **Conformance** - Similar to compliance for functionality, but this characteristic relates to portability.
- **Replaceability** - The plug and play aspect of software components, that is how easy is it to exchange a given software component within a specified environment.

What are best coding practices

- Machine Readable Vs Human Readable code
- Self-documented code
- Shorter the sweeter
- Let's be afraid of the DEAD code
- You can see my code's flow without looking at it – logging
- Modifications should not break the existing code – Junits
- Can it perform better – paying the technical debt.

The Pillars of OOPs

- **Abstraction** – You should know WHAT I am doing but not HOW I am doing it
 - Coding to interfaces, provide facades, using access specifiers
- **Encapsulation** – Lets provide behaviour while hiding properties
 - Writing class with states (private) and behaviour (public)
- **Inheritance** – Lets generalize and specialize entities close to real world
 - Set the family hierarchy
- **Polymorphism** – Thou shall exist in many forms

SOLID Design Principles

SOLID Design Principles

S.O.L.I.D is an abbreviation for the first five object-oriented design(OOD) principles by **Robert C. Martin**, popularly known as **Uncle Bob**.

- **S** – Single Responsibility Principle
- **O** – Open Close Principle
- **L** – Liskov's Substitution Principle
- **I** – Interface Segregation Principle
- **D** – Dependency Inversion (not Injection)

Single Responsibility Principle (SRP)

Every object should have a single responsibility, and that responsibility should be entirely encapsulated by the class.

- A class should have only one reason to change
- If we have two reasons to change a class, we should split it into two classes
- Modifications to a functionality in a class having multiple responsibilities, might affect the other functionalities of that class

Single Responsibility Principle

```
class Customer {  
    public void Add() {  
        try {  
            // Database code goes here  
        }  
        catch (Exception ex) {  
            System.IO.File.WriteAllText(@"c:\Error.txt", ex.ToString());  
        }  
    }  
}
```

The above customer class is doing things **WHICH HE IS NOT SUPPOSED TO DO.**



Single Responsibility Principle

The famous swiss knife.

Too many responsibilities on a single thing can cause problems.



```
class FileLogger {  
    public void Handle(string error) {  
        System.IO.File.WriteAllText(@"c:\Error.txt", error);  
    }  
}
```

```
class Customer {  
    private FileLogger obj = new FileLogger();  
    Public virtual void Add() {  
        try { // code goes here  
        }  
        catch (Exception ex) {  
            obj.Handle(ex.ToString());  
        }  
    }  
}
```

Single Responsibility Principle

```
public class UserService {  
    public void Register(string email, string password) {  
        if (!email.Contains("@"))  
            throw new ValidationException("Email is not an email!");  
        var user = new User(email, password);  
        _database.Save(user);  
        _smtpClient.Send(new MailMessage("abc@xyz.com", email){Subject="Hello  
fool!"});  
    }  
}
```

Solution

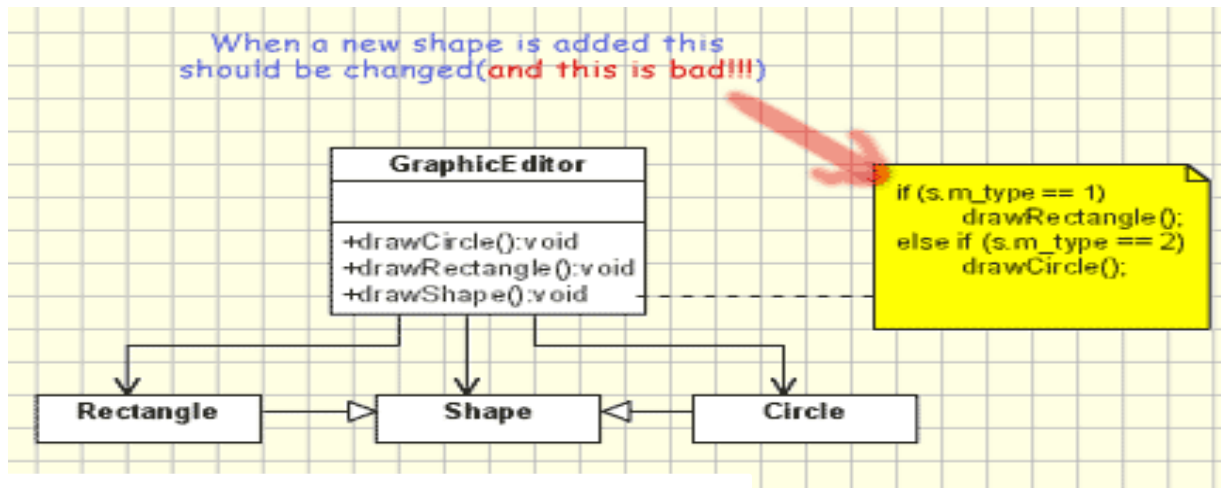
```
public class UserService{  
    public void Register(string email, string password){  
        if (!ValidateEmail(email))  
            throw new ValidationException("Email is not an email!");  
        var user = new User(email, password);  
        _database.Save(user);  
        _smtpClient.Send(new MailMessage("abc@xyz.com", email){Subject="Hello  
fool!"});  
    }  
    public bool ValidateEmail(string email){  
        return email.Contains("@");  
    }  
}
```

Open Close Principle

- Software entities like **classes**, **modules** and **functions** should be open for extension but closed for modifications
- The design and writing of the code should be done in a way that new functionality should be added with minimum changes in the existing code
- The design should be done in a way to allow the adding of new functionality as new classes, keeping as much possible the existing code unchanged



Open Close Principle



// Open-Close Principle - Bad example

```
class GraphicEditor {
    public void drawShape(Shape s)
    {
        if (s.m_type==1)
            drawRectangle(s);
        else if (s.m_type==2)
            drawCircle(s);
    }
    public void drawCircle(Circle r) {....}

    public void drawRectangle(Rectangle r)
    {....}
}
```

```
class Shape {
    int m_type;
}
class Rectangle extends Shape {
    Rectangle() {
        super.m_type=1;
    }
}
class Circle extends Shape {
    Circle() {
        super.m_type=2;
    }
}
```

Open Close Principle

// Open-Close Principle - Good example

```
class GraphicEditor {  
    public void drawShape(Shape s)  
{  
        s.draw();  
    }  
}  
  
abstract class Shape {  
    abstract void draw();  
}
```

```
class Rectangle extends Shape {  
    public void draw() {  
        // draw the rectangle  
    }  
}  
  
class Circle extends Shape {  
    public void draw() {  
        // draw the rectangle  
    }  
}
```


Interface Segregation Principle

- Clients should not be forced to implement interfaces they don't use
- Problem – FAT interfaces
- Instead of one fat interface many small interfaces are preferred based on groups of methods, each one serving one submodule



INTERFACE SEGREGATION PRINCIPLE

You Want Me To Plug This In, Where?

Interface Segregation Principle

//Bad example

```
public interface IMachine {  
    public void print();  
    public void staple();  
    public void scan();  
    public void photoCopy();  
}
```

```
public class XeroxMachine implements IMachine {  
    @Override  
    public void print()  
        { System.out.println("Printing Job"); }  
    @Override  
    public void staple()  
        { System.out.println("Stapling Job"); }  
    @Override  
    public void scan()  
        { System.out.println("Scan Job"); }  
    @Override public void photoCopy()  
        { System.out.println("Photo Copy"); }  
}
```

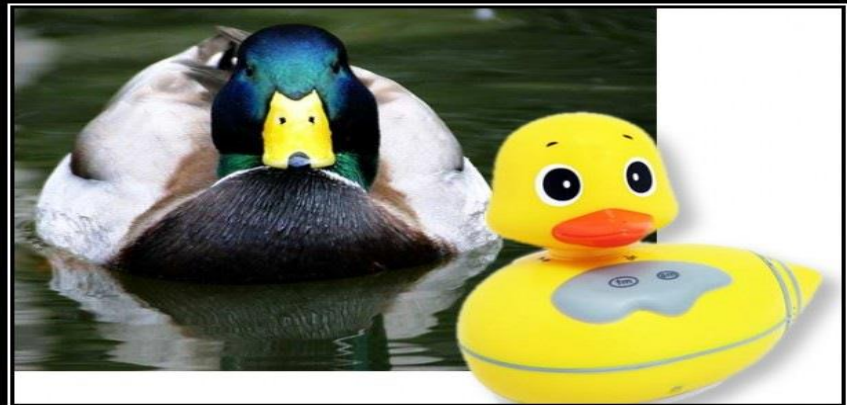
Interface Segregation Principle

```
public interface IPrinter {  
    public void print();  
}  
  
public interface IScanner {  
    public void fax();  
}  
  
public interface IStapler {  
    public void staple();  
}  
  
public interface IPhotoCopier  
{  
    public void photoCopy();  
}
```

- New Machine implements – IPrinter, IScanner and IPhotoCopier
- Individual machines can implement individual interfaces
- The old IMachine interface could still implement all four interfaces and be used the same way as previously

Liskov's Substitution Principle (LSP)

- Written by **Barbara Liskov** in 1988.
- Methods that use references to the base classes must be able to use the objects of the derived classes without knowing it
- The subtypes must be replaceable for the super type references without affecting the program execution
- This means - **make sure that new derived classes are extending the base classes without changing their behaviour**



LSKOV SUBSTITUTION PRINCIPLE

If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

How can we identify LSP violation?

- Derived class may require less functionalities than the Base class, so some methods would be redundant
- We might be using IS-A to check for Super-Sub relationships, but LSP also requires that the Sub types must be substitutable for the Super class.

How can we identify LSP violation?

// Bad example

```
class Bird {
    public void fly(){...}
    public void eat(){...}
}

class Crow extends Bird {}
class Ostrich extends Bird{
    fly(){
        throw new
        UnsupportedOperationException();
    }
}
```

```
public BirdTest{
    public static void main(String[]
    args){
        List<Bird> birdList = new
        ArrayList<Bird>();
        birdList.add(new Bird());
        birdList.add(new Crow());
        birdList.add(new Ostrich());
        letTheBirdsFly ( birdList );
    }

    static void letTheBirdsFly (
    List<Bird> birdList ){
        for ( Bird b : birdList )
            { b.fly(); }
    }
}
```

How can we identify LSP violation?

```
class Bird {  
    public void eat() {...}  
}
```

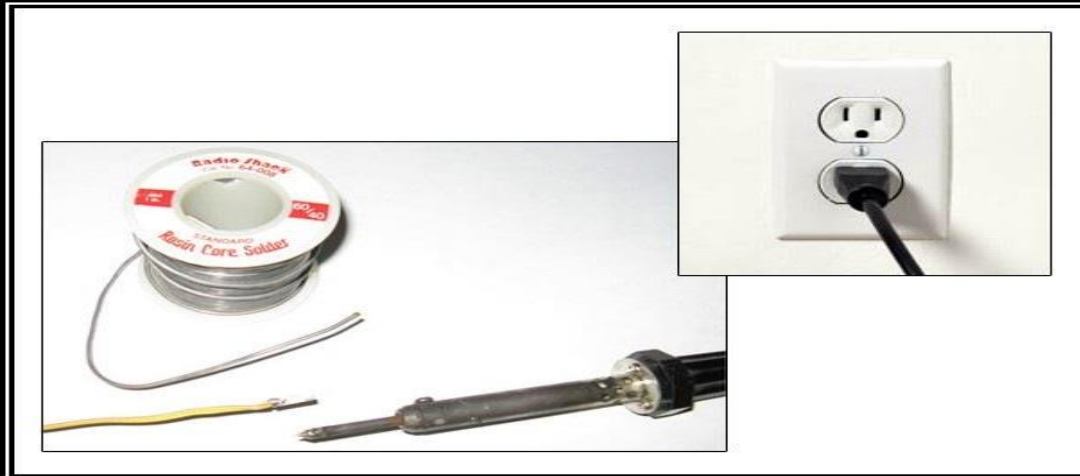
```
class FlightBird extends Bird {  
    public void fly() {...}  
}
```

```
class NonFlight extends Bird { }
```

Dependency Inversion Principle

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.

High Level Classes --> Abstraction Layer <-- Low Level Classes



DEPENDENCY INVERSION PRINCIPLE

Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

Dependency Inversion Principle

// Dependency Inversion Principle - Bad example

```
class Worker {  
    public void work() { // ....working }  
}
```

```
class Manager {  
    Worker worker;  
    public void setWorker(Worker w) { worker = w; }  
    public void manage() { worker.work(); }  
}
```

```
class SuperWorker {  
    public void work() { //.... working much more }  
}
```

Dependency Inversion Principle

// Dependency Inversion Principle - Good example

```
interface IWorker {  
    public void work();  
}  
class Worker implements IWorker{  
    public void work() { // ....working }  
}  
class SuperWorker implements IWorker{  
    public void work() { //.... working much  
more }  
}  
class Manager {  
    IWorker worker;  
    public void setWorker(IWorker w) {  
worker = w; }  
    public void manage() { worker.work(); }  
}
```

- High level classes not work directly with low level classes
- they use interfaces as an abstract layer
- instantiation of new low level objects inside the high level classes should not be done using **new**
- Instead we use Creational design patterns - Factory Method, Abstract Factory, Prototype
- Dependency Injection and also IoC containers help us in DIP

SOLID Summary

Principle Name	What it says?	howtodoinjava.com
Single Responsibility Principle	One class should have one and only one reasonability	
Open Closed Principle	Software components should be open for extension, but closed for modification	
Liskov's Substitution Principle	Derived types must be completely substitutable for their base types	
Interface Segregation Principle	Clients should not be forced to implement unnecessary methods which they will not use	
Dependency Inversion Principle	Depend on abstractions, not on concretions	

Pillars of Code Quality & Code Metrics

Pillars of Code Quality

- **Modularity**

- The functional independence of program components.

- **Maintainability**

- Effort required to locate & fix an error in a program.

- **Extendibility**

- The degree to which architectural, data or procedural design can be extended.

- **Testability**

- Effort required to test a program to ensure it perform its intended function.

- **Reusability**

- Extent to which a program (or parts of a program) can be reused in other applications.

- **Performance**

- The amount of computing resources and code required by a program to perform its function.

Why do we need code metrics?

- Metrics give a measure of code quality.
- Code conforming to benchmarks of metrics is easier to maintain.
- Performance of the software is directly impacted by poor code quality
- The effective strategy is to achieve quality by
 - periodically assessing the complexity metrics
 - Refactoring to keep them under acceptable limits

Code Metrics Description

■ Size

- Lines of Code (LOC)
- Number Of Operations (NOO)

■ Complexity

- Cyclomatic Complexity (CC)
- Maximum Number Of Levels (MNOL)
- Depth of Inheritance Hierarchy (DOIH)
- Weighted Methods Per Class¹ (WMPC-1)
- Comment Ratio (CR)
- Response For Class (RFC)
- Maximum Number of Parameters (MNOP)

■ Modularity

- Coupling Between Objects (CBO)
- Polymorphism – Number of Added Methods (NOAM)

Recommended Standard

Item	Description	Updated Upper Limit	Updated Lower Limit
DOIH	Depth of inheritance hierarchy	5	-
NOO	Number of operations	17	-
NOAM	Number of added methods	18	-
CBO	Coupling between objects	13	-
Method Max. CC	Cyclomatic Complexity at class level	10	-
CR	Comment Ratio	-	5
LOC	Lines of Code	700	-
MNOL	Max number of levels	5	-
RFC	Response for class	53	-
WMPC1	Weighted methods per class 1	40	-
MNOP	Max number of parameters	4	-

Code Metrics - Size

- Lines of Code (LOC)
 - A measure of the number of code lines & indicates size of a class. This is the traditional measure of size.
- Number of Operations (NOO)
 - Counts the number of local methods in a class

Corrective Actions - Size

- **Extract SuperClass**: Generates new class using existing classes. It means creating an ancestor class from several operations of a given class or from several different classes
- If a class has a high number of operations, it might be wise to consider whether it would be appropriate to divide it into subclasses.
- **Pull Up Operation**: copies an operation from a subclass to a super-class, deleting the original and optionally changing its visibility

Cyclomatic Complexity (CC)

- Counts the number of possible paths through an algorithm by counting the number of *if*, *for* and *while* statements in the operation's body.
- Indicates the number of linearly independent paths through a program module.
- A measure of the number of tests required to execute all code paths through a procedure.

Cyclomatic Complexity (CC)

```
Public Function YesNoFlagToString(ByVal v_sYNFlag As String) As String
    Dim sResult As String
    If v_sYNFlag = "Y" Then
        sResult = "Yes"
    Else
        sResult = "No"
    End If
    YesNoFlagToString = sResult
End Function
```

Cyclomatic complexity = 2

- Path 1
- Path 2

Cyclomatic Complexity (CC)

- Corrective Action

- Checks & Loops should be minimized and nested loops should be avoided as far as possible.
- Extract Operation: It means extracting a semantically complete piece of code into a new method.

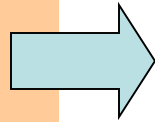
Maximum Number Of Levels (MNOL)

- A measure of the number of levels of nesting in your method or class. Counts the maximum depth of if, for and while branches in the bodies of methods.
- Simplify Operation: Conditions in loops can be combined to reduce nesting

Maximum Number Of Levels (MNOL)

Conditions in loops can be combined to reduce nesting.

```
package PackageA;
public class Shape {
    public Shape() {
    }
    public void
    printColor(int iColor) {
        if(iColor==RED){
            print ("PRIMARY");
        }else if(iColor==BLUE){
            print ("PRIMARY");
        } else
        if(iColor==YELLOW){
            print ("PRIMARY");
        }
        else{ print ("SECONDARY");
        }
    }
}
```



```
package PackageA;
public class Shape
{
    public Shape() {
    }
    public void printColor(int
    iColor)
    {
        if(iColor==RED || iColor==BLUE
        || iColor==YELLOW){
            print ("PRIMARY");
        }else {
            print ("SECONDARY");
        }
    }
}
```

Depth of Inheritance Hierarchy (DOIH)

- Counts how far down the inheritance hierarchy a class or interface is declared.
- Corrective Action
 - DOIH is corrected by Inheritance Based Refactoring.
Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure
 - Merge class hierarchy vertically if size of class (NOO, CC, WMPC1) is small

Weighted Methods Per Class 1(WMPC-1)

- Sum of the complexity of all methods for a class, where each method is weighted by its cyclomatic complexity.
- The number of methods and the complexity of the methods involved is a predictor of how much time and effort is required to develop and maintain the class
- Corrective Action
 - Extract SuperClass: Generates new class using existing classes. It means creating an ancestor class from several operations of a given class or from several different classes.
 - Split Classes-according to functionality or business modules

Response For Class (RFC)

- Includes methods in the class's inheritance hierarchy and methods that can be invoked on other objects.
- A class that provides a larger response set is considered to be more complex and require more testing efforts than one with a smaller overall design complexity.
- Corrective Action
 - Divide the class into subclasses
 - Extract Operation: It means extracting a semantically complete piece of code into a new method
 - Extract Interface: Generates new Interface using existing classes. This will improve the reusability of the class
 - Split Classes-according to functionality or business modules.

Max Number of Parameters (MNOP)

- Counts the highest number of parameters defined for a single operation, from among all the operations in the class.
- Methods with many parameters tend to be more specialized and so are less likely to be reusable
- Has a negative impact on Maintainability & Extendibility
- Testing of such functions having higher number of parameters needs more effort.
- Flexibility of signature in case of enhancements will involve more efforts
- Corrective Action
 - Put parameters in logical objects or structures.
 - Minimize the functionality in methods

Coupling Between Objects (CBO)

- Represents the number of other classes to which a class is coupled.
- Counts the number of reference types that are used in attribute declarations, formal parameters, return types, throws declarations and local variables, and types from which attribute and method selections are made.

Code Metrics - Polymorphism

- Number of Added Methods (NOAM)
 - Counts the number of operations added by a class. Inherited and overridden operations are not counted.
 - A large value of this parameter means that the child class has very little in common with the parent class.

Code Metrics - Polymorphism

- Impact on Code Quality
 - A high value of NOAM has a negative impact on
 - Testability: A high NOAM indicates that the class is very specialized. So more effort will be required to test it.
- Corrective Action
 - Check if the class should really be inherited from the parent
 - Break down the class into several smaller classes if possible instead of inheritance

Static Factory Methods

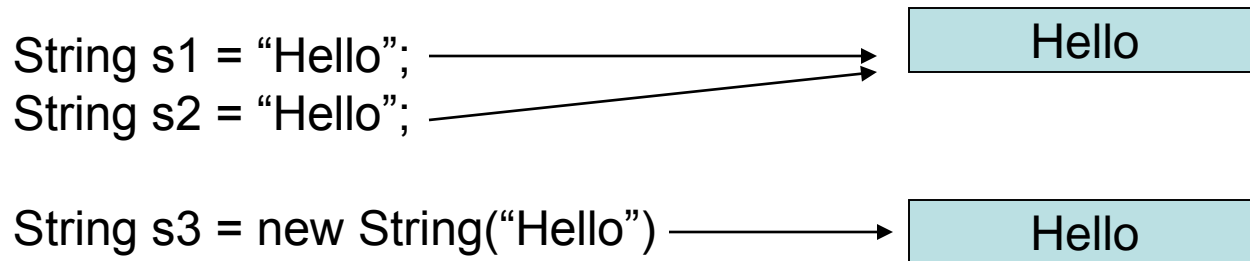
- Consider providing static factory methods instead of constructors
 - They can have meaningful names
 - They are not required to create new objects every time they are invoked
 - They can return any type of objects
 - They can be used to implement singletons

```
class Line
{
    public Line() { }
    public Line(char letter) { }
}
```

```
class Line
{
    public static createLine() { }
    public createLineWith (char letter) { }
}
```

Duplicate Objects

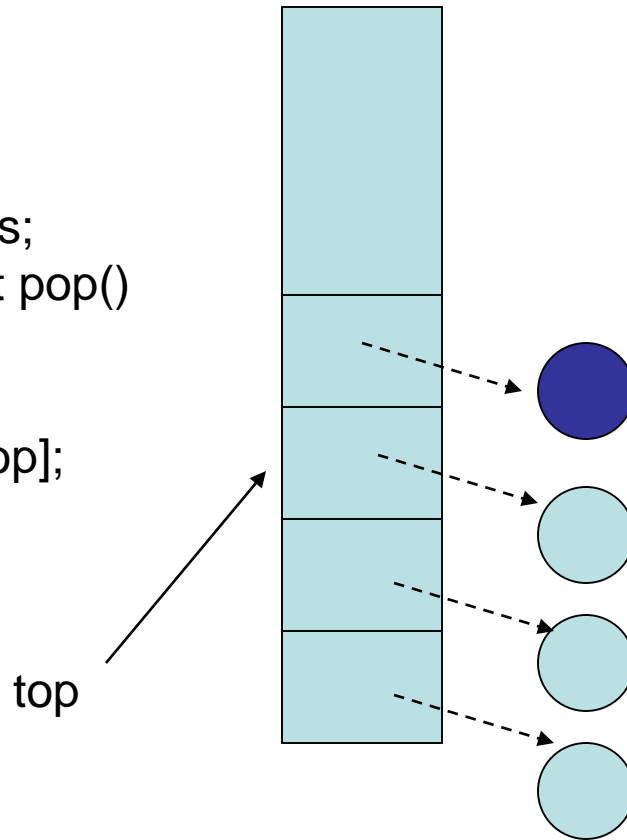
- Avoid creating duplicate objects
 - Use String literals instead of new String()
- Use private static fields and static initializers to reuse internal temporary objects



Obsolete References

- Eliminate obsolete object references
 - Assign null to objects when they are no longer needed
 - E.g. assigning null to stack element in pop()

```
class Stack
{
  Element[] items;
  public Element pop()
  {
    top--;
    return items[top];
  }
}
```



```
class Stack
{
  Element[] items;
  public Element pop()
  {
    top--;
    Element temp=items[top];
    items[top]=null;
    return temp;
  }
}
```

Minimize accessibility

- Minimize accessibility of classes and members
 - Should be private, package-private, protected and public in that order of preference
 - Improves the principle of encapsulation
 - Reduces coupling between modules

Immutability

- An immutable class is a class whose instances cannot be modified.

E.g. String

- They are easier to design, implement and use
 - Less error prone, thread-safe and secure
- Rules
 - Don't provide mutator methods
 - Use final methods and fields
 - Make fields private
 - Make defensive copies in constructors and accessors

Composition over Inheritance

- Favor composition over inheritance
 - Create a wrapper around a class to be extended in functionality
 - It is safer to use inheritance only within a package
 - Inheritance breaks encapsulation
 - Reduces the Depth of Inheritance

```
class Engine
{
    void start() { }
```

```
Class Car extends Engine
{
}
```

```
// Car car = new Car();
// car.start()
```

```
class Engine
{
    void start() { }
```

```
Class Car
{
    Engine engine;
    void start() { engine.start(); }
```

```
// Car car = new Car();
// car.start()
```

Methods

- Check parameters for validity before use
- Make defensive copies when needed before passing objects to methods
- Choose method names carefully
- Avoid long parameter list
- User overloading judiciously: the choice of which method will be invoked is done at compile time
- Return zero length arrays instead of nulls
- Variables
 - Declare where it is first used
 - Every declaration should have initializer

Exceptions

- Use Exceptions only for exceptional conditions instead of flow control
- Use checked exceptions for recoverable conditions and runtime exceptions for programming errors
- Catch exceptions only when you can take corrective action otherwise propagate the same to the caller
- Never write empty catch blocks
 - At least use `printStackTrace()`
- Document all exceptions thrown by a method

Unit Testing Best Practices

Unit Testing Best Practices

- Use a standard framework like **Junit**
- Aim for unit tests that are
 - Extremely fast
 - Extremely reliable – tests that fail are worse than no test
- Should run completely in-memory
 - Avoid HTTP requests, database access, filesystem
 - Instead use functional test for these
- Do not skip unit tests, rather remove them from source control
- To skip temporarily, use @Ignore annotation

JUnit

- Open-source framework, developer centric, extensible.
- For writing unit tests and running collections of tests.
- Provides a simple way to explicitly test specific areas of a Java program.
- Can be used to test a hierarchy of program code either singularly or as multiple units.
- JUnit in version 4.x is a test framework which uses annotations to identify methods that specify a test.

Philosophy

- Promotes the idea of first testing then coding
 - It is possible to setup test data for a unit which defines what the expected output is and then code until the tests pass.
 - It is believed by some that this practice of "**test a little, code a little, test a little, code a little...**" increases programmer productivity and stability of program code whereas reducing programmer stress and the time spent debugging.

Benefits

- Faster code development.
- Better quality code.
- Simple to use.
- Tests check their own results.
- Provides immediate feedback.
- Tests can be composed into a test suite.
- Increase stability of application.

Junit Annotations

Use Annotations

- **@Test** - To check the expected result of the code execution versus the actual result. It is use an assert method, provided by the JUnit or another assert framework. These method calls are typically called asserts or assert statements.
- **@RunWith** – Use to specify a runner object that the test case will be called from
- **@Before** – use to tag a method to be called before a test case. This usually being used to initialise data for the test case
- **@After** – use to tag a method to be called after a test case. This is usually used to do clean up processes

Junit Annotations and Description

- **@Test public void method()** - The @Test annotation identifies a method as a test method.
- **@Test (expected = Exception.class)** - Fails if the method does not throw the named exception.
- **@Test(timeout=100)** - Fails if the method takes longer than 100 milliseconds.
- **@Before public void method()** - This method is executed before each test. It is used to prepare the test environment (e.g., read input data, initialize the class).
- **@After public void method()** - This method is executed after each test. It is used to cleanup the test environment (e.g., delete temporary data, restore defaults).
- **@BeforeClass - public static void method()** - This method is executed once, before the start of all tests. It is used to perform time intensive activities, for example, to connect to a database.
- **@AfterClass public static void method()** - This method is executed once, after all tests have been finished. It is used to perform clean-up activities, for example, to disconnect from a database.

Assert statements

- **fail(message)** - Let the method fail. Might be used to check that a certain part of the code is not reached or to have a failing test before the test code is implemented.
- **assertTrue([message,] boolean condition)** - Checks that the boolean condition is true.
- **assertFalse([message,] boolean condition)** - Checks that the boolean condition is false.
- **assertEquals([message,] expected, actual)** - Tests that two values are the same. Note: not the content of the arrays.
- **assertNull([message,] object)** - Checks that the object is null.
- **assertNotNull([message,] object)** - Checks that the object is not null.
- **assertSame([message,] expected, actual)** - Checks that both variables refer to the same object.
- **assertNotSame([message,] expected, actual)** - Checks that both variables refer to different objects.

Code coverage

- Code coverage highlights the amount of code that is covered by the tests that are carried out on the code.
- JUnit – To write test cases
- Emma – To verify how much code is covered by test cases

Demo

- 1. JUnit Simple Example
- 2. JUnit Annotations
- 3. JUnit assertions

Introduction to PMD/CPD

- PMD is a tool which helps identifying duplicate code segments (Copy/Paste Detector)
- Duplicated code means that bugs can appear in several places.
- It means longer compilation times and bigger binaries.
- It means major pain for whoever has to go through the program and make changes.
- Why it happens?
 - Programmers are in a hurry (copy/paste is quick!)
 - Projects merge resulting in duplicate code
 - In large projects, Programmers forget what is already available
- Solution? – Refactoring
 - Locate duplicate code,
 - Move it into a method,
 - Add a parameter to the method,
 - Update the original code to use the new method, and
 - Test the original code

PMD Features

- PMD/CPD scans Java source code and looks for potential problems like:
 - Empty try/catch/finally/switch blocks
 - Unused local variables, parameters and private methods
 - Empty if/while statements
 - Overcomplicated expressions - unnecessary if statements, for loops that could be while loops
 - Classes with high Cyclomatic Complexity measurements
- Runs a set of static code analysis rules on some Java source code files and generates a list of problems found

Ready to use Rulesets

- Basic JSF rules
- Basic JSP rules
- Basic rules
- Braces rules
- Code size rules
- Design rules
- Import statement rules
- J2EE rules
- JavaBean rule
- JUnit rules
- Java logging rules
- Optimization rules
- Strict Exception rules
- String and StringBuffer rules
- Security code guidelines
- Type resolution rules
- Unused code rules

Current RuleSet - <http://pmd.sourceforge.net/pmd-4.3.0/rules/index.html>

Basic Rules Examples

- **EmptyCatchBlock** - Empty Catch Block finds instances where an exception is caught, but nothing is done. In most circumstances, this swallows an exception which should either be acted on or reported
- **JumbledIncrementer** - Avoid jumbled loop incrementers - it's usually a mistake, and it's confusing even if it's what's intended
- **ForLoopShouldBeWhileLoop** - Some for loops can be simplified to while loops - this makes them more concise
- **UnnecessaryConversionTemporary** - Avoid unnecessary temporaries when converting primitives to Strings

Basic Rules Examples

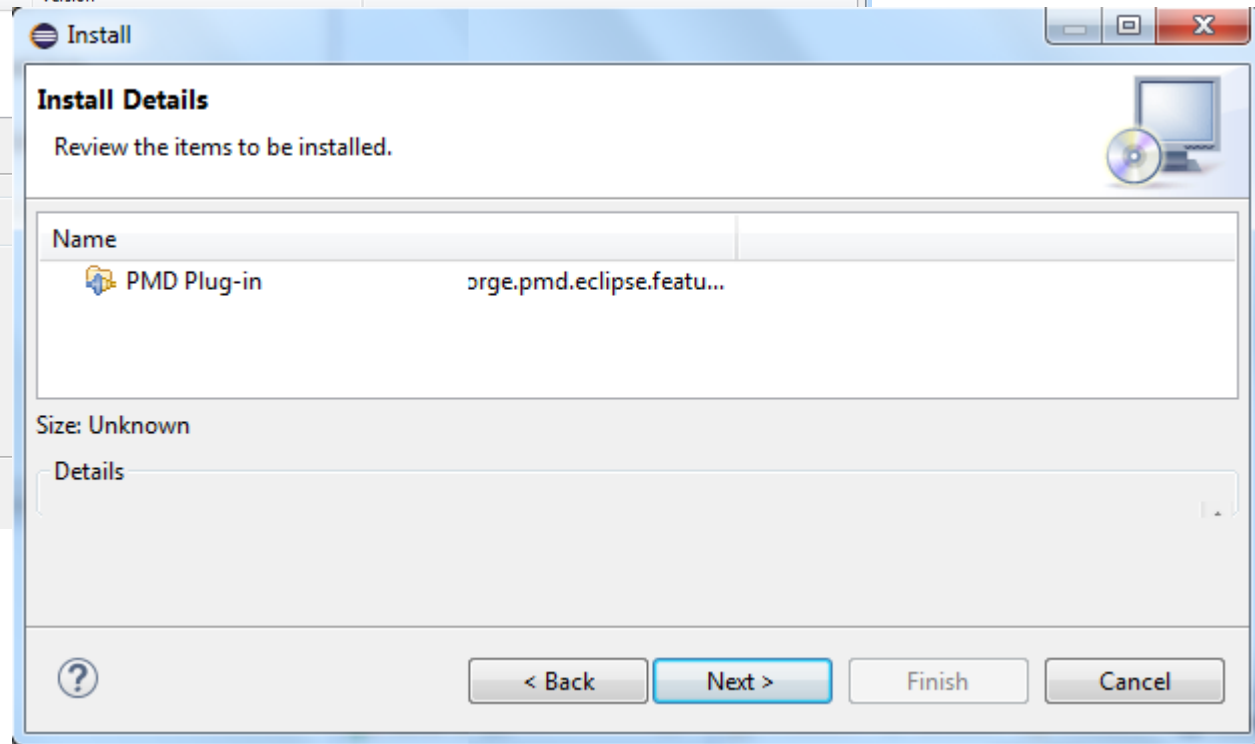
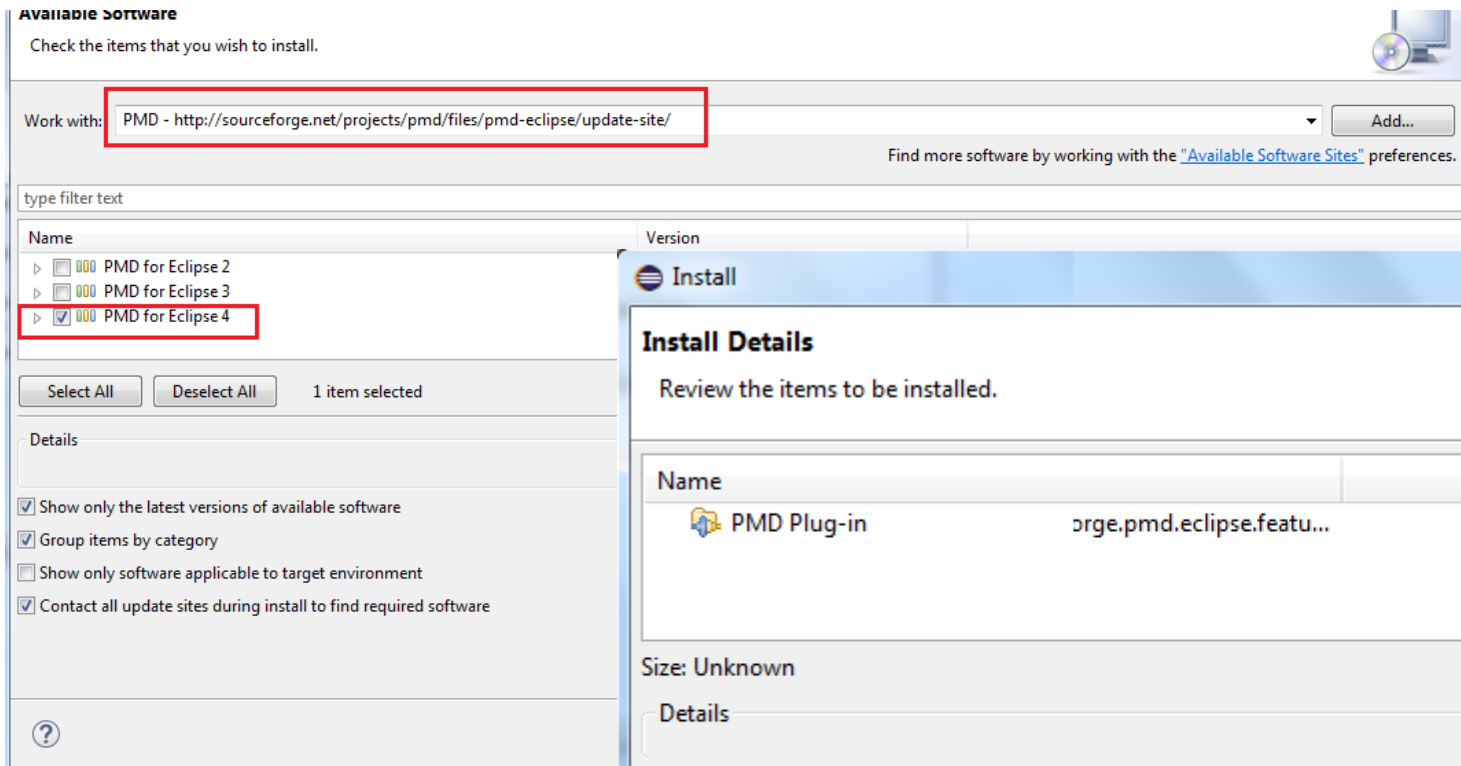
- **OverrideBothEqualsAndHashCode** -Override both public boolean `Object.equals(Object other)`, and public int `Object.hashCode()`, or override neither
- **ReturnFromFinallyBlock** - Avoid returning from a finally block - this can discard exceptions
- **EmptyStaticInitializer** -An empty static initializer was found
- **UnconditionalIfStatement** - Do not use "if" statements that are always true or always false
- **BooleanInstantiation** - Avoid instantiating Boolean objects; you can reference `Boolean.TRUE`, `Boolean.FALSE`, or call `Boolean.valueOf()` instead

Basic Rules Examples

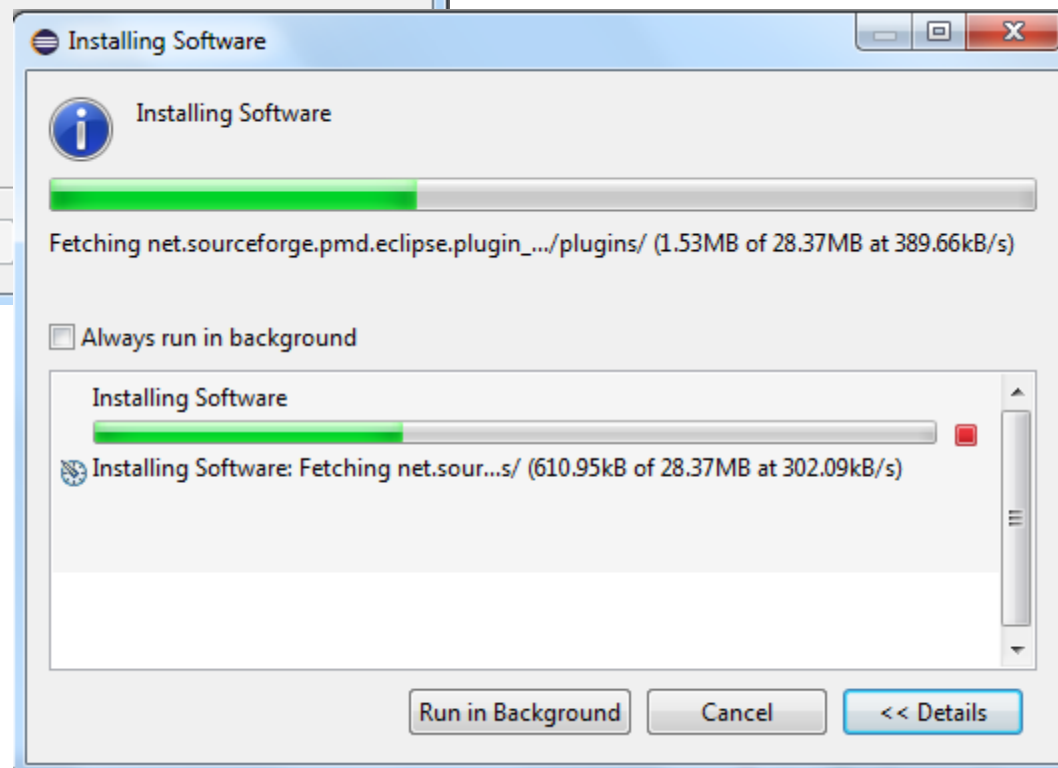
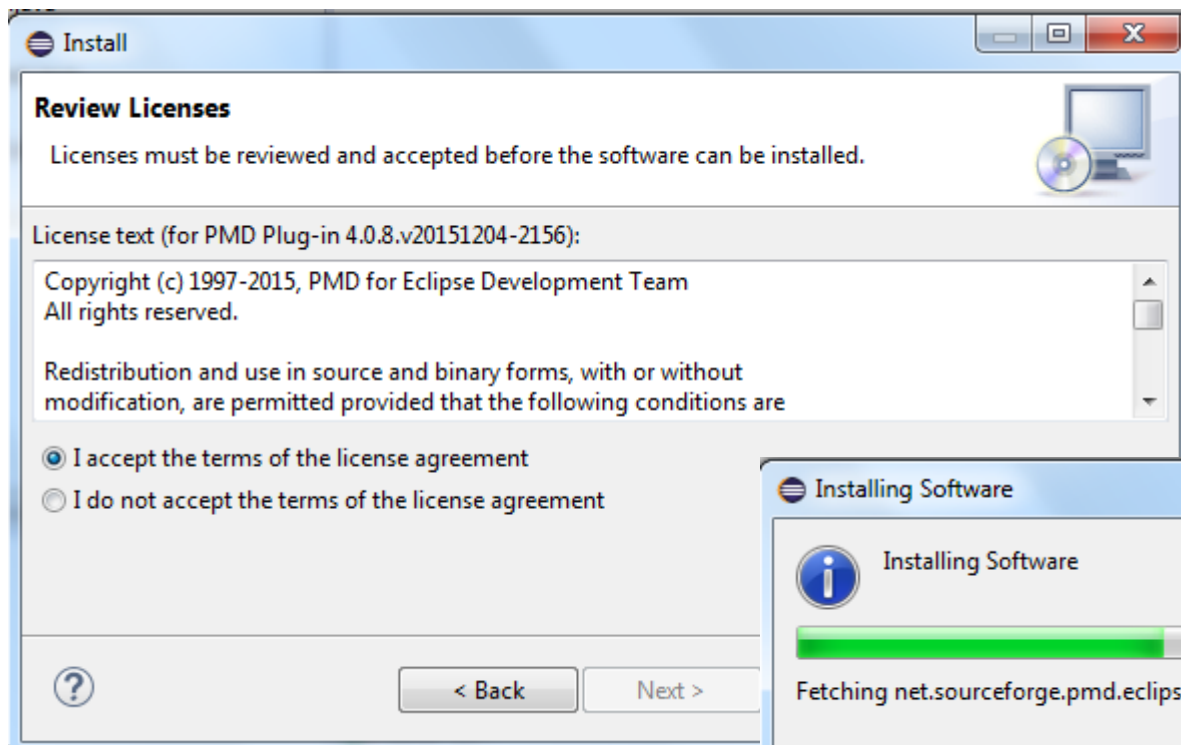
- **UselessOperationOnImmutable** - An operation on an Immutable object (String, BigDecimal or BigInteger) won't change the object itself. The result of the operation is a new object. Therefore, ignoring the operation result is an error.
- **AvoidThreadGroup** - Avoid using `java.lang.ThreadGroup`; although it is intended to be used in a threaded environment it contains methods that are not thread safe.
- **BrokenNullCheck** - The null check is broken since it will throw a `NullPointerException` itself.

Eclipse Plug-in

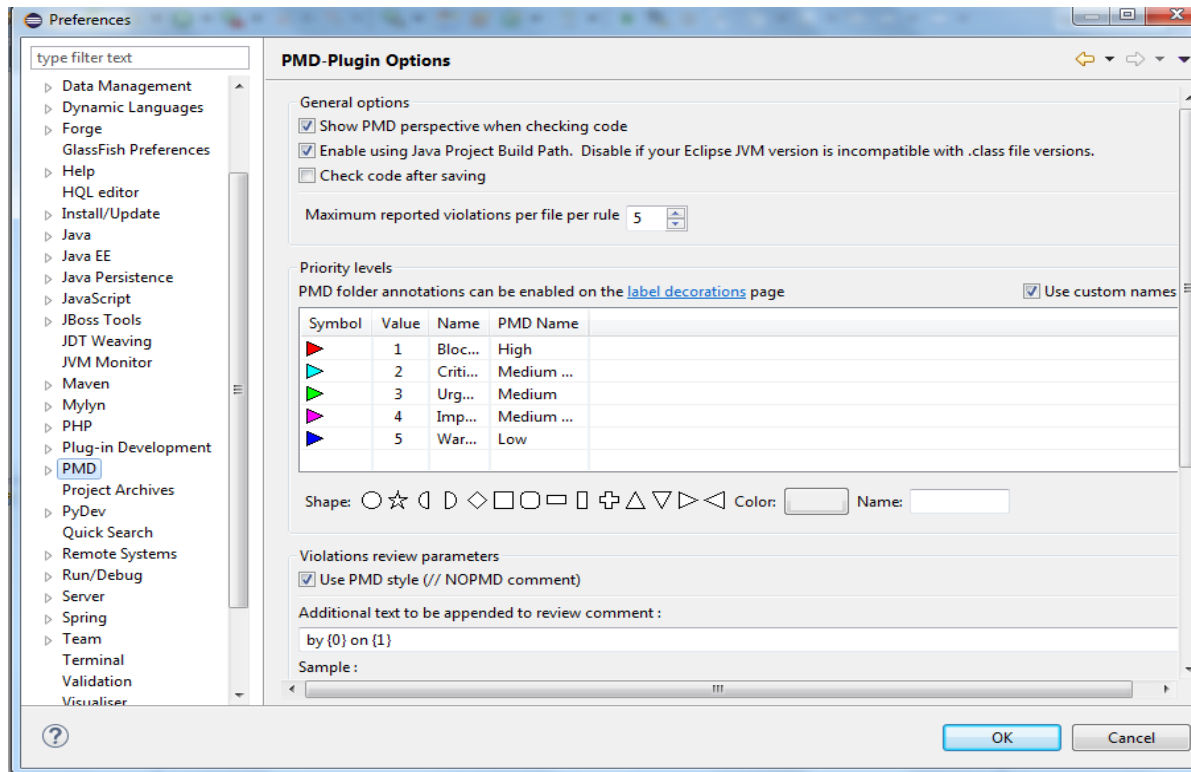
1. Plugin URL - <http://sourceforge.net/projects/pmd/files/pmd-eclipse/update-site/>
2. Current RuleSet - <http://pmd.sourceforge.net/pmd-4.3.0/rules/index.html>



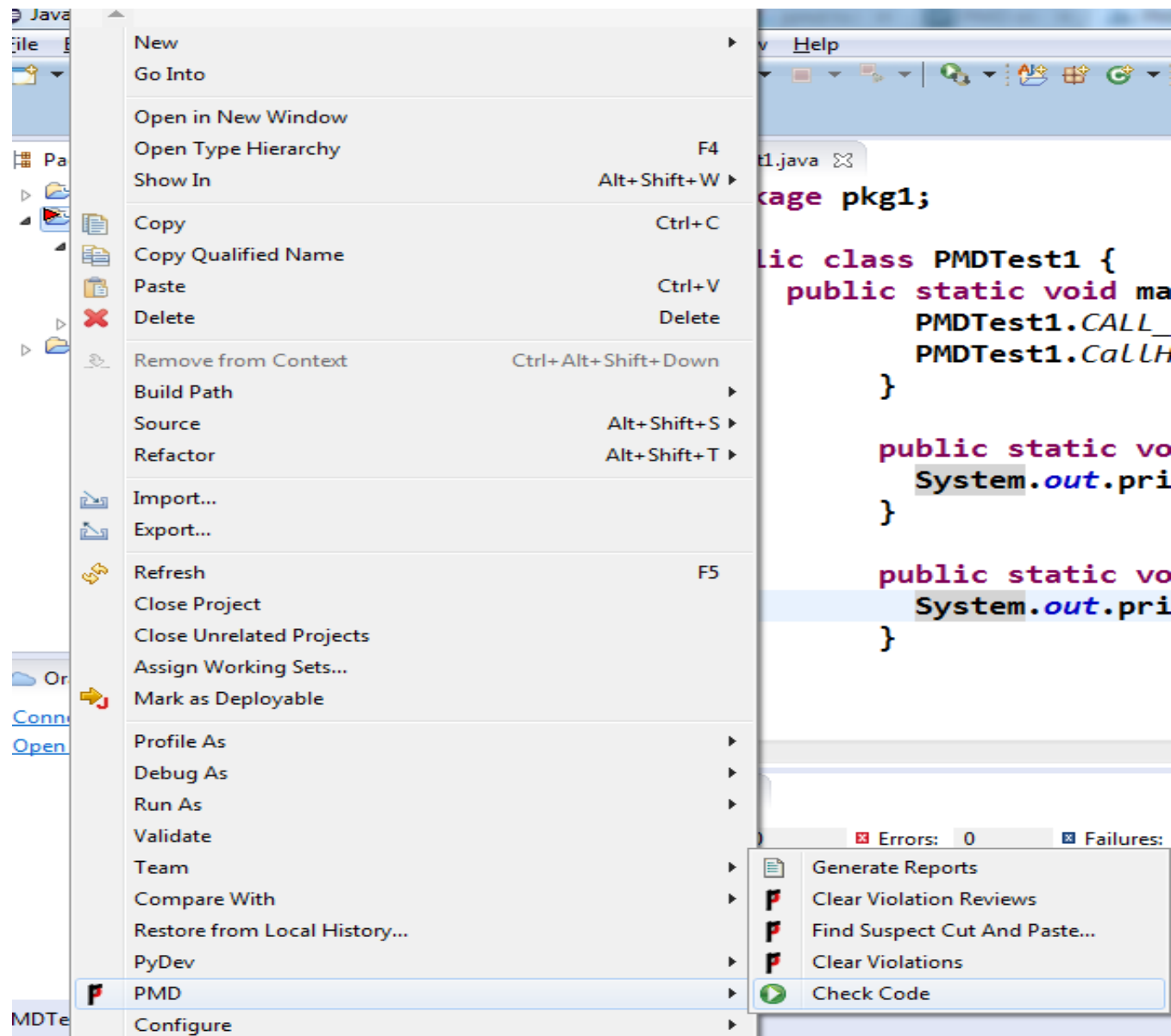
Eclipse Plug-in



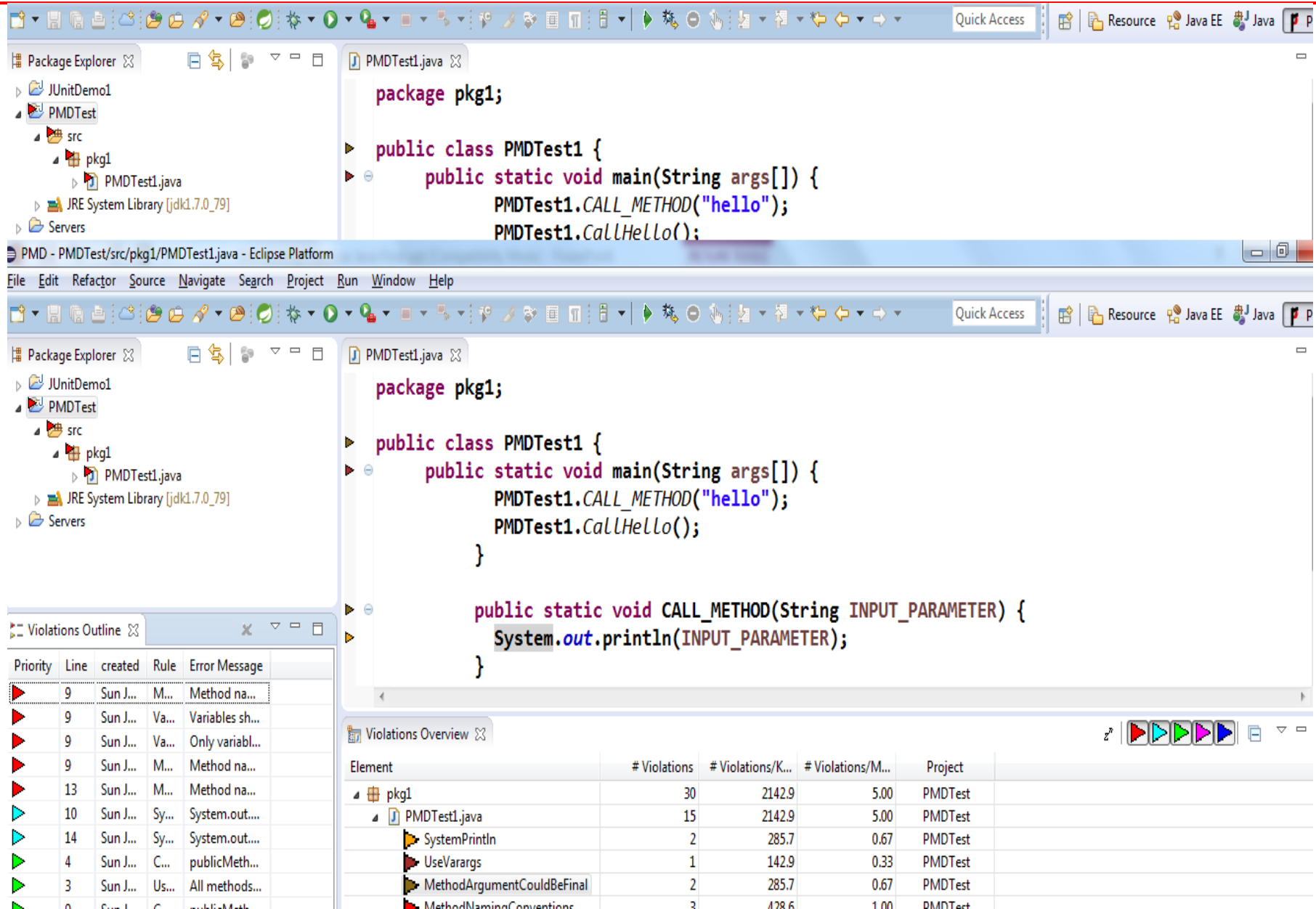
Check PMD Plug-in in EClipse



Run PMD



Running PMD



The screenshot displays the Eclipse IDE interface with the PMD plugin. The Package Explorer on the left shows the project structure: JUnitDemo1, PMDTest, and src/pkg1/PMDTest1.java. The main editor shows the source code of PMDTest1.java, which includes a package declaration, a class PMDTest1, and two methods: a main method and a static method CALL_METHOD. The PMD tool has identified 30 violations across the project, with 15 violations in PMDTest1.java. The violations are listed in the Violations Outline and Violations Overview panels.

Violations Outline

Priority	Line	created	Rule	Error Message
9	9	Sun J...	M...	Method na...
9	9	Sun J...	Va...	Variables sh...
9	9	Sun J...	Va...	Only variabl...
9	9	Sun J...	M...	Method na...
13	13	Sun J...	M...	Method na...
10	10	Sun J...	Sy...	System.out...
14	14	Sun J...	Sy...	System.out...
4	4	Sun J...	C...	publicMeth...
3	3	Sun J...	Us...	All methods...

Violations Overview

Element	# Violations	# Violations/K...	# Violations/M...	Project
pkg1	30	2142.9	5.00	PMDTest
PMDTest1.java	15	2142.9	5.00	PMDTest
SystemPrintln	2	285.7	0.67	PMDTest
UseVarargs	1	142.9	0.33	PMDTest
MethodArgumentCouldBeFinal	2	285.7	0.67	PMDTest
MethodNamingConventions	2	172.6	1.00	PMDTest

Logging Best Practices

Logging Best Practices

- Use the appropriate tools for the job – SLF4J, Log4j, Logback, Perf4J
- Use logging levels appropriately
 - ERROR, WARN, INFO, DEBUG, TRACE
- What to log
- Avoid side effects
- Be concise and descriptive
- Define the logging pattern
 - `<pattern>%d{HH:mm:ss.SSS} %-5level [%thread][%logger{0}] %m%n</pattern>`
- **Download URL -**
<http://logging.apache.org/log4j/2.x/download.html>

Logging Best Practices

- Use the appropriate tools for the job
 - `log.info("Happy and carefree logging");`
 - `log.debug("Found {} records matching filter: '{}'", records, filter);`
 - `log.debug("Found " + records + " records matching filter: " + filter + "");`
- Don't forget, logging levels are there for you
 - `ERROR, WARN, INFO, DEBUG, TRACE`
- Do you know what you are logging?
 - `log.debug("Processing request with id: {}", request.getId());`
 - `log.debug("Returning users: {}", users);`
- Avoid side effects
 - ```
try {
 log.trace("Id=" + request.getUser().getId() + " accesses " +
 manager.getPage().getUrl().toString())
} catch (NullPointerException e) {}
```

# Logging Best Practices

---

- Be concise and descriptive
  - `log.debug("Message processed");`
  - `log.debug(message.getJMSMessageID());`
  - `log.debug("Message with id '{} processed", message.getJMSMessageID());`
- Tune your pattern
  - ```
<appender name="STDOUT"
class="ch.qos.logback.core.ConsoleAppender">
  <encoder>
    <pattern>%d{HH:mm:ss.SSS} %-5level
[%thread][%logger{0}] %m%n</pattern>
  </encoder>
</appender>
```

Logging Best Practices

- Log method arguments and return values

- ```
public String printDocument(Document doc, Mode mode) {
 log.debug("Entering printDocument(doc={}, mode={})", doc,
mode);
 String id = //Lengthy printing operation
 log.debug("Leaving printDocument(): {}", id);
 return id;
}
```

- Watch out for external systems

- ```
Collection<Integer> requestIds = //...  
if(log.isDebugEnabled())  
    log.debug("Processing ids: {}", requestIds);  
else  
    log.info("Processing ids size: {}", requestIds.size());
```

- Log exceptions properly
- Logs easy to read, easy to parse

Logging Best Practices

- Log exceptions properly

- try {
 Integer x = null;
 ++x;
} catch (Exception e) {
 log.error(e); //A
 log.error(e, e); //B
 log.error("" + e); //C
 log.error(e.toString()); //D
 log.error(e.getMessage()); //E
 log.error(null, e); //F
 log.error("", e); //G
 log.error("{} ", e); //H
 log.error("{} ", e.getMessage()); //I
 log.error("Error reading configuration file: " + e); //J
 log.error("Error reading configuration file: " + e.getMessage());
//K
 log.error("Error reading configuration file", e); //L
}

Logging Best Practices

- Logs easy to read, easy to parse
 - `log.debug("Request TTL set to: {} ({})", new Date(ttl), ttl);`
 - `// Request TTL set to: Wed Apr 28 20:14:12 CEST 2010 (1272478452437)`
 - `final String duration = DurationFormatUtils.formatDurationWords(durationMillis, true, true);`
 - `log.info("Importing took: {}ms ({})", durationMillis, duration);`
 - `//Importing took: 123456789ms (1 day 10 hours 17 minutes 36 seconds)`

Logger statements – the performance impact

- Logger statements are among the most heavily used statements in most Java applications.
- While the code contains Logger statement at different levels, the common practice is to set a default log level for the entire application.
- This will ensure that only important statements are logged by default.

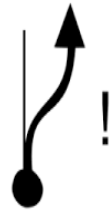
Logger statements – the performance impact

```
import java.util.logging.Level;
import java.util.logging.Logger;
public class LoggerTest {
    private static Logger logger = Logger.getLogger("LoggerTest");
    public static void main(String args[]){
        // set the default log level to INFO
        logger.setLevel(Level.INFO);
        long startTime = System.nanoTime();
        logger.finer(getLogMessage());    // This should not print anything
        long endTime = System.nanoTime();
        System.out.println("Time for Logger without Boolean check: "+(endTime - startTime));
        startTime = System.nanoTime();
        if (false)
        {
            logger.finer(getLogMessage());
        }
        endTime = System.nanoTime();
        System.out.println("Time for Logger with Boolean check: "+(endTime - startTime));
    }
    private static String getLogMessage()
    {
        return "Lets understand the "+"performance impact of Java Logger.";
    }
}
```

- 1 Time for Logger without Boolean check: 13440 ns
- 2 Time for Logger with Boolean check: 640 ns

Legacy Code Refactoring

- Refactoring and not rewriting (unless really required)
- Identify code to be refactored
- Take a diversion approach
- Try to modularise the old code
- Add/modify unit tests as per the new code
- Regression, functional, integration tests



Collections Best Practices

Collections Best Practices

Interface	Duplicates?	Implementations					Historical
Set	no	HashSet	...	LinkedHashSet	...	TreeSet	...
List	yes	...	ArrayList	...	LinkedList	...	Vector , Stack
Map	no duplicate keys	HashMap	...	LinkedHashMap	...	TreeMap	Hashtable , Properties

Collections Best Practices

- The best general purpose or primary implementations are **ArrayList**, **LinkedHashMap** and **LinkedHashSet**
- Use these unless any specific feature like **sorting** or **ordering** is required
- Use highest level abstraction for object specification wherever possible
- Code your methods to use interfaces rather than implementations
 - **List myData = new ArrayList()**
- Avoid using legacy implementation – **Vector**, **Hashtable**

Collections Best Practices

Principle features of non-primary implementations

- **HashMap** slightly better performance than **LinkedHashMap**, undefined iteration order
- **HashSet** slightly better performance than **LinkedHashSet**, undefined iteration order
- **TreeSet** and **TreeMap** ordered and sorted, but slower
- Keys of a Map and Items in a Set are highly recommended to be immutable objects as these should not change their state
- **LinkedList** has fast adding to the start of the list, and fast deletion from the interior via iteration

Collections Best Practices

Iteration order for above implementations:

- **HashSet** - undefined
- **HashMap** - undefined
- **LinkedHashSet** - insertion order
- **LinkedHashMap** - insertion order of keys (by default), or 'access order'
- **ArrayList** - insertion order
- **LinkedList** - insertion order
- **TreeSet** - ascending order, according to Comparable / Comparator
- **TreeMap** - ascending order of keys, according to Comparable / Comparator

For **LinkedHashSet and **LinkedHashMap**, the re-insertion of an item does not affect insertion order.**

Collections Best Practices

▪ Arrays vs Collections

- Arrays are by far the fastest but least flexible implementation of a **collection**
- Use an array only if you **must** access the elements via a static array index, and only if you expect the array order and content to be immutable

Collections Best Practices

- Sets should be used when
 - You have a set of objects without duplicates
 - More concerned about existence of element than order
- The iterator will return values but not in any guaranteed order
- Use SortedSet to return elements in natural (alphabetic or numeric)

Collections Best Practices

- Use Map if you want a lookup or mapping
- Do not use the old Hashtable
- Add key-value mapping using the put(key, value) method
- To traverse through mappings first get keys using keyset() method
- To go through keys in natural sorted order, wrap the keys in a TreeSet

Collections Best Practices

```
Map myMap = new HashMap() ;  
myMap.put("id", new Integer(12345)) ;  
myMap.put("name", "John Doe") ;
```

```
Set keys = new TreeSet(myMap.keySet()) ;  
Iterator i = keys.iterator() ;  
while (i.hasNext()) {  
    Class key = (Class) i.next() ;  
    Class value = (Class) myMap.get(key) ;  
    System.out.println("key : value - " + key + " : " + value)  
}
```

Generics

Collections Best Practices

```
Set group = new HashSet() ;  
group.add("b") ;  
group.add("c") ;  
group.add("c") ;  
Iterator i = group.iterator() ;  
while (i.hasNext()) {  
    // Casting always necessary  
    Class object = (Class) i.next();  
    System.out.println(object.toString()) ;  
}
```

```
Set orderedGroup = new TreeSet() ;  
orderedGroup.add("b") ;  
orderedGroup.add("c") ;  
orderedGroup.add("c") ;  
Iterator i = orderedGroup.iterator() ;  
while (i.hasNext()) {  
    // Casting always necessary  
    Class object = (Class) i.next();  
    System.out.println(object.toString()) ;  
}
```

Generics

- Generics has following benefits
 - Stronger type-checking at compile time
 - Fending off ClassCastException that might be thrown at runtime
 - Cleaner code by eliminating explicit casts

The most commonly used type parameter names are:

- E – Element (used extensively by the Java Collections Framework, for example ArrayList, Set etc.)
- K – Key (Used in Map)
- N – Number
- T – Type
- V – Value (Used in Map)

Generics

```
public class Box<T> {  
    private T t;  
    public void add(T t) {  
        this.t = t;  
    }  
    public T get() {  
        return t;  
    }  
    public static void main(String[] args) {  
        Box<Integer> integerBox = new Box<Integer>();  
        Box<String> stringBox = new Box<String>();  
        integerBox.add(new Integer(10));  
        stringBox.add(new String("Hello World"));  
        System.out.printf("Integer Value :%d\n\n", integerBox.get());  
        System.out.printf("String Value :%s\n", stringBox.get());  
    }  
}
```

Enum

Enums

- Introduced in Java 1.5 as a new type
- Fields consist of fixed set of constants
- Better than normal
- constant fields in Java classes
- For enum constants, equals and == amount to same thing
- Enum constants are implicitly public static final

Enums

```
public enum ThreadStates {  
    START,  
    RUNNING,  
    WAITING,  
    DEAD;  
}
```

```
public class ThreadStatesConstant {  
    public static final int START = 1;  
    public static final int WAITING = 2;  
    public static final int RUNNING = 3;  
    public static final int DEAD = 4;  
}
```

```
private static void simpleEnumExample(ThreadStates th) {  
    if(th == ThreadStates.START)  
        System.out.println("Thread started");  
    else if (th == ThreadStates.WAITING)  
        System.out.println("Thread is waiting");  
    else if (th == ThreadStates.RUNNING)  
        System.out.println("Thread is running");  
    else System.out.println("Thread is dead");  
}
```

```
private static void simpleConstantsExample(int i) {  
    if(i == ThreadStatesConstant.START)  
        System.out.println("Thread started");  
    else if (i == ThreadStatesConstant.WAITING)  
        System.out.println("Thread is waiting");  
    else if (i == ThreadStatesConstant.RUNNING)  
        System.out.println("Thread is running");  
    else System.out.println("Thread is dead");  
}
```

```
private static void enumOverConstants()  
{  
    //Enum values are fixed  
    simpleEnumExample(ThreadStates.START);  
    simpleEnumExample(ThreadStates.WAITING);  
    simpleEnumExample(ThreadStates.RUNNING);  
    simpleEnumExample(ThreadStates.DEAD);  
    simpleEnumExample(null);  
  
    simpleConstantsExample(1);  
    simpleConstantsExample(2);  
    simpleConstantsExample(3);  
    simpleConstantsExample(4);  
    //we can pass any int constant  
    simpleConstantsExample(5);  
}
```

Enums

- We can pass any int constant to the simpleConstantsExample method but we can pass only fixed values to simpleEnumExample, so it provides type safety
- We can change the int constants value in ThreadStatesConstant class but the above program will not throw any exception and program might not work as expected but if we change the enum constants, we will get compile time exception which removes any possibility of runtime issues.

Difference between Iterator,
ListIterator and Enumeration ?

Multithreading

Introduction to Multithreading

Why Multithreading?

- Make the UI more responsive
- Take advantage of multiprocessor systems
- Simplify modeling in simulation application
- Perform asynchronous or background processing

Multithreading & Multitasking

- **Two types of Multitasking:**
 - Thread based
 - Process based
- A Process is a program which is under execution
- Process based multitasking allows to execute two or more programs concurrently

Multithreading & Multitasking

- In process based multitasking, a **program** is the smallest unit of code that can be dispatched by the scheduler
- In thread based multitasking, a **thread** is the smallest unit of code that can be dispatched by the scheduler
- This means that a single program can have two or more tasks which can be executed simultaneously
- **Multithreading** – Thread based Multi Tasking

What is a Thread?

- A Thread is an independent, concurrent path of execution through a program
- Threading is a facility to allow multiple activities to execute simultaneously within a single process
- Sometimes referred to as **lightweight processes**
- Every process has at least one thread - the *main* thread

Multithreading Example

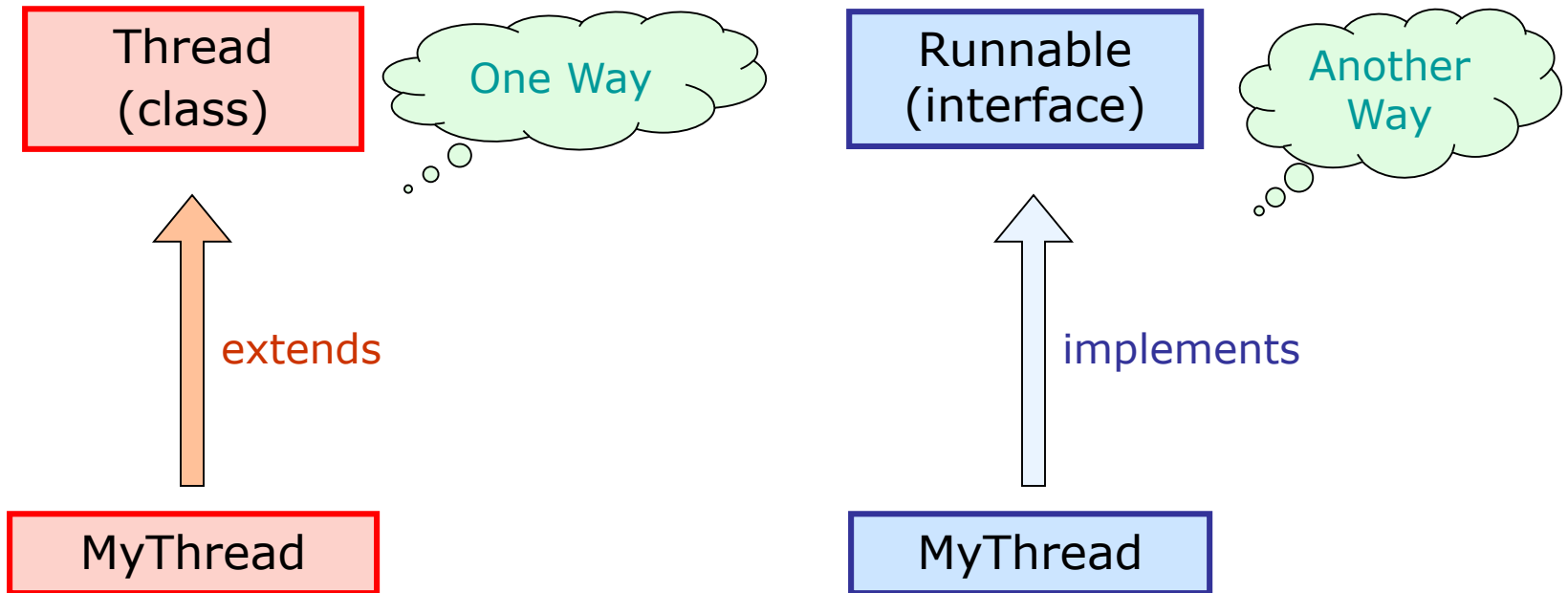
- **Consider your basic word processor**

- You have just written a large amount of text in MS Word editor and now hit the save button
- It takes a noticeable amount of time to save new data to disk, this is all done with a separate thread in the background
- Without threads, the application would appear to hang while you are saving the file and be unresponsive until the save operation is complete

Implementing Multithreading

Two ways:

- Extending the **Thread** Class
- Implementing the **Runnable** Interface



Extending the *Thread* Class

- Override the run() method in the subclass from the Thread class to define the code executed by the thread

```
public class ThreadExample extends Thread
{
    private String data;

    public ThreadExample(String data) {
        this.data = data;
    }

    public void run() {
        System.out.println("I am a thread with "+data);
    }
}
```

Running Threads

- Create an instance of this subclass (ThreadExample)
- Invoke the *start()* method on the instance of the class to make the thread eligible for running

```
public class ThreadExampleMain
{
    public static void main(String[] args) {
        Thread myThread = new ThreadExample("my data");
        myThread.start();
    }
}
```


Using the *Runnable* Interface

- Why this approach is required?
- Implement the Runnable interface
- Override the run() method to define the code executed by thread

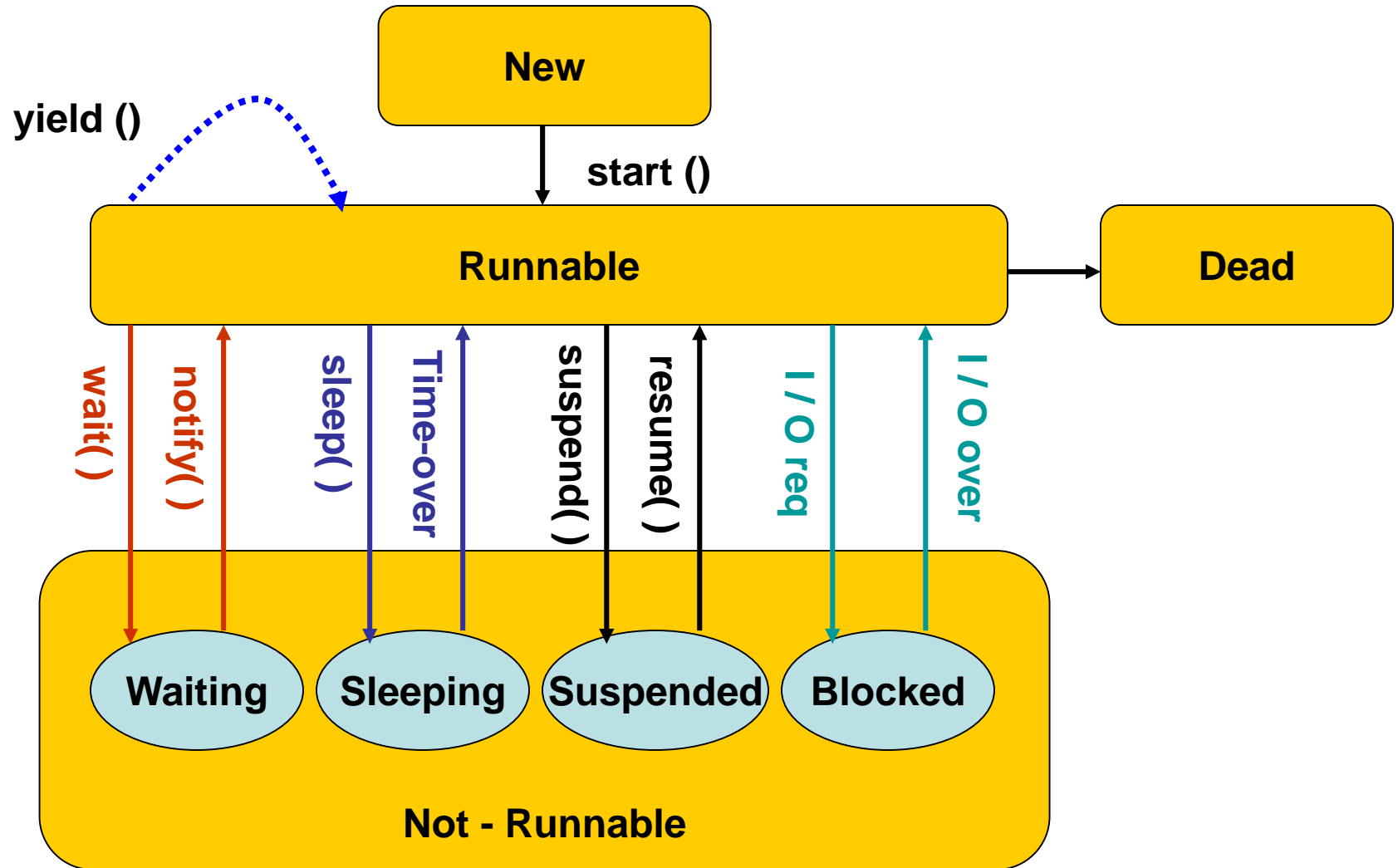
```
public class RunnableExample extends SomeClass
    implements Runnable
{
    private String data;
    public RunnableExample(String data) {
        this.data = data;
    }
    public void run() {
        System.out.println("I am a thread: "+data);
    }
}
```

Using the *Runnable* Interface (Contd...)

- Create an object of Thread class
- Invoke the start() method on the instance of the Thread class

```
public class RunnableExampleMain
{
    public static void main(String[] args) {
        RunnableExample myRunnableObject = new
            RunnableExample("my data");
        Thread myThread = new Thread(myRunnableObject);
        myThread.start();
    }
}
```

Thread Life Cycle



Thread Life Cycle (Contd...)

A Thread object has the following states in its lifecycle:

Born	The thread object has been created
Ready / Runnable	The thread is ready for execution
Running	The thread is currently running
Blocked	The thread is blocked for some operation (e.g. I/O Operations)
Sleeping	The thread is not utilizing its time slice till the timer elapses
Suspended	The thread is not utilizing its time slice till <i>resume()</i> is called
Waiting	Thread enters into waiting on calling <i>wait()</i> method
Dead	The thread has finished execution or aborted (The dead thread cannot be started again)

Using *sleep()*, *yield()*

- Once a thread gains control of the CPU, it will execute until one of the following occurs:
 - Its *run()* method exits
 - A higher priority thread becomes *runnable* & pre-empts it
 - Its time slice is up (on a system that supports time slicing)
 - It calls *sleep()* or *yield()*

yield()	the current thread paused its execution temporarily and has allowed other threads to execute
sleep()	the thread sleeps for the specified number of milliseconds, during which time any other thread can use the CPU

Using *join()*

- A call to the *join* method on a specific thread causes the current thread to block until that thread is completed

```
public class ThreadExampleMain
{
    public static void main(String[] args) {
        Thread myThread = new ThreadExample("my data");
        myThread.start();
        System.out.println("I am the main thread");

        myThread.join();
        System.out.println("waiting for myThread");
    }
}
```

Other *Thread* Operations

- **interrupt()** method
 - Interrupts the thread on which it is invoked
- **interrupted()** method
 - Returns a boolean value indicating the interrupted status of the current thread & resets the same to not interrupted
- A combination of these two methods is useful for requesting a thread to yield, sleep or terminate

Thread Priorities

- The JVM chooses which thread to run according to a **fixed priority algorithm**
- Every thread has a priority between the range of ***Thread.MIN_PRIORITY(1)*** and ***Thread.MAX_PRIORITY(10)***
- By default a thread is instantiated with the same priority as that of the thread that created it
- Thread priority can be changed using the ***setPriority()*** method of the Thread class

Thread Priorities

- Thread priority can be obtained using the ***getPriority()*** method of the Thread class
- Threads with higher priorities are run to completion before Threads with lower priorities get a chance of CPU time
- The algorithm is *preemptive*, so if a lower priority thread is running, and a higher priority thread becomes runnable, the high priority thread will pre-empt the lower priority thread

Synchronization

- Sometimes, multiple threads may be accessing the same resources concurrently
 - Reading and / or writing the same file
 - Modifying the same object / variable
- Synchronization controls thread execution order
- Synchronization eliminates data races
- Java has built in primitives to facilitate this coordination

Synchronization

▪ **Producer / Consumer Example:**

- There are 2 threads, a producer & a consumer, both accessing the same object of type **CustomerBill**
- **CustomerBill** is a simple object that holds a single value as its contents
- The producer thread randomly generates values and stores them in the **CustomerBill** object
- The consumer then retrieves these values as they are generated by the producer

Synchronization: Producer

```
public class Producer extends Thread
{
    private CustomerBill bill;
    public Producer(CustomerBill b) {
        bill = b;
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            bill.put(i);
            try {
                sleep((int) (Math.random() * 100));
            } catch (InterruptedException e) { }
        }
    }
}
```

Synchronization: Consumer

```
public class Consumer extends Thread
{
    private CustomerBill bill;
    public Consumer(CustomerBill b) {
        bill= b;
    }
    public void run() {
        int value = 0;
        for (int i = 0; i < 10; i++) {
            value = bill.get();
        }
    }
}
```

Synchronization: Problems

- **When the producer produces a value, it stores it in the **CustomerBill** , and then the consumer is only supposed to retrieve it, once and only once**
- Depending on how the threads are scheduled:
 - The producer could produce two values before the consumer is able to retrieve one
 - The consumer could consume the same value twice before the producer has produced the next value
- If the producer & consumer access the **CustomerBill** at the same time, they could produce an inconsistent state or miss a produced value

Synchronization: Solution

```
public class CustomerBill
{
    private int contents;
    private boolean available = false;

    public synchronized int get()
    { ... }

    public synchronized void put(int value)
    { ... }
}
```

- When a thread executes a synchronized method, it locks the object of the method
- No synchronized methods can be called by another thread on that object until it is unlocked

Synchronization: Wait / Notify

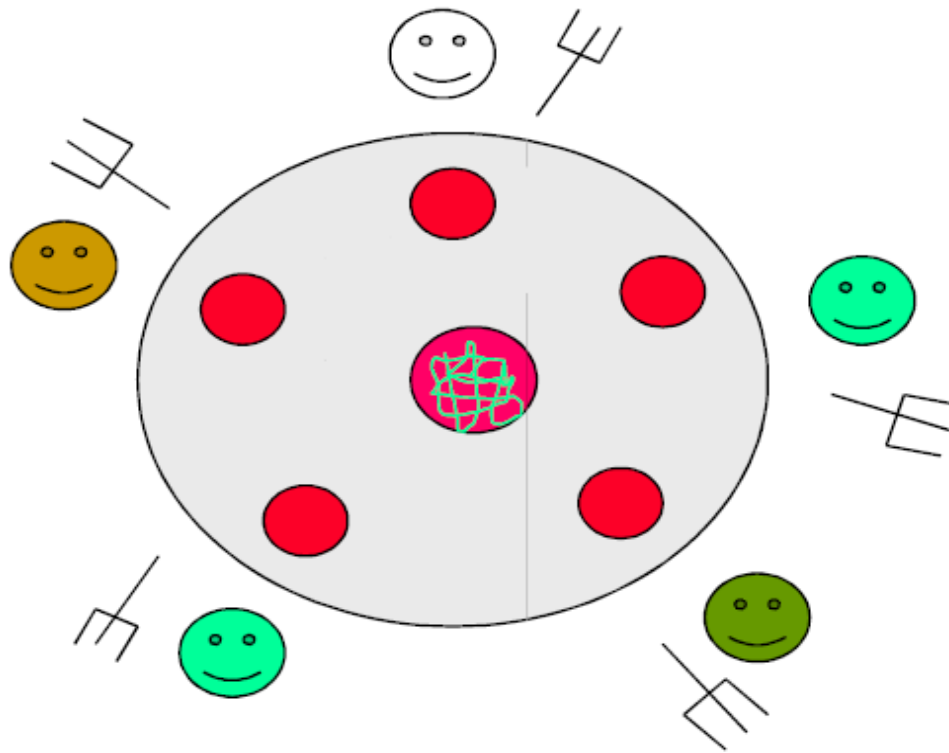
- **Synchronized** methods stop the producer & consumer from modifying the **CustomerBill** at the same time

```
public synchronized int get() {  
    while (available == false) {  
        try {  
            //wait for Producer to put value  
            wait();  
        } catch (InterruptedException e) { }  
    }  
    available = false;  
    //notify Producer that value has been retrieved  
    notifyAll();  
    return contents;  
}
```


Thread Deadlock

The Dining Philosopher Problem

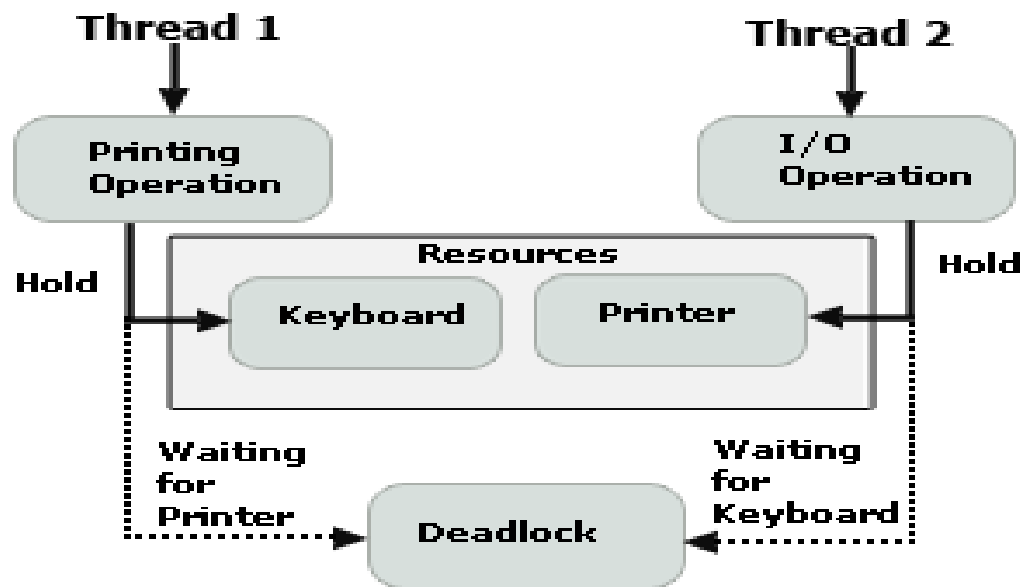
- All philosophers are holding a single fork & are waiting for each other



- Java multithreading has a provision to solve this problem

Thread Deadlock (Contd...)

- If a thread is waiting for an object lock held by the second thread
- The second thread is waiting for an object lock held by the first one
- Example: 2 threads having printing & I/O operations respectively at a time
 - Thread1 needs a printer which is held by Thread2
 - Thread2 needs the keyboard which is held by Thread1



Concurrency

Concurrency Utilities

- Enables development of simple yet powerful multithreaded applications
 - Like Collection provides rich data structure handling capability
- Beat C performance in high-end server applications
- Provide richer set of concurrency building blocks
 - *wait()*, *notify()* and *synchronized* are too primitive
- Enhance scalability, performance, readability and thread safety of Java applications

Why Use Concurrency Utilities?

- Reduced programming effort
- Increased performance
- Increased reliability
 - Eliminate threading hazards such as deadlock, starvation, race conditions, or excessive context switching are eliminated
- Improved maintainability
- Increased productivity

Concurrency Utilities

- Task Scheduling Framework
- Callable's and Future's
- Synchronizers
- Concurrent Collections
- Atomic Variables
- Locks
- Nanosecond-granularity timing

Task Scheduling Framework

- **Executor/Executorservice/Executors** framework supports
 - standardizing submission
 - scheduling
 - execution
 - control of asynchronous tasks according to a set of execution policies
- **Executor** is an interface
- **Executorservice** extends Executor
- **Executors** is factory class for creating various kinds of **Executorservice** implementations

Executor Interface

- **Executor** interface provides a way of de-coupling task **submission** from the **execution**
 - submission is standardized
 - execution: mechanics of how each task will be run, including details of thread use, scheduling – captured in the implementation class
- Example

```
Executor executor = getSomeKindofExecutor();  
executor.execute(new RunnableTask1());  
executor.execute(new RunnableTask2());
```
- Many **Executor** implementations impose some sort of
- limitation on how and when tasks are scheduled

Executor and ExecutorService

- **ExecutorService** adds lifecycle management

```
public interface Executor {
```

```
void execute(Runnable command);
```

```
}
```

```
public interface ExecutorService extends Executor {
```

```
void shutdown();
```

```
List<Runnable> shutdownNow();
```

```
boolean isShutdown();
```

```
boolean isTerminated();
```

```
boolean awaitTermination(long timeout, TimeUnit unit);
```

```
// other convenience methods for submitting tasks
```

```
}
```

Executor and ExecutorService

- **ExecutorService** adds task submission methods

```
public interface Executor {  
    void execute(Runnable command);  
}  
  
public interface ExecutorService extends Executor {  
    void shutdown();  
    List<Runnable> shutdownNow();  
    boolean isShutdown();  
    boolean isTerminated();  
    boolean awaitTermination(long timeout,  
        TimeUnit unit);  
    // other convenience methods for submitting tasks  
    <T> Future<T> submit(Callable<T> task);  
    Future<?> submit(Runnable task);  
    <T> Future<T> submit(Runnable task, T result);  
}
```

Concurrent Collections

BlockingQueue Interface

- Provides thread safe way for multiple threads to manipulate collection
- **ArrayBlockingQueue** is simplest concrete implementation
- Full set of methods
 - **put()**
 - **offer()** [non-blocking]
 - **peek()**
 - **take()**
 - **poll()** [non-blocking and fixed time blocking]

Concurrency: Atomic Variables

Atomics

- **java.util.concurrent.atomic**
 - Small toolkit of classes that support lock-free threadsafe programming on single variables

```
AtomicInteger balance = new AtomicInteger(0);  
public int deposit(integer amount) {  
return balance.addAndGet(amount);  
}
```

Concurrency: Locks

Locks

▪ Lock interface

- More extensive locking operations than synchronized block
- No automatic unlocking – use try/finally to unlock
- Non-blocking access using **tryLock()**

▪ ReentrantLock

- Concrete implementation of Lock
- Holding thread can call **lock()** multiple times and not block
- Useful for recursive code

ReadWriteLock

- Has two locks controlling read and write access
 - Multiple threads can acquire the read lock if no threads have a write lock
 - If a thread has a read lock, others can acquire read lock but nobody can acquire write lock
 - If a thread has a write lock, nobody can have read/write lock
 - Methods to access locks

`rwl.readLock().lock();`

`rwl.writeLock().lock();`

Annotations

Annotations

- Annotations are used to affect the way programs are treated by tools and libraries
- Annotations are used by tools to produce derived files
 - Tools: Compiler, IDE, Runtime tools
 - Derived files : New Java code, deployment descriptor, class files

Annotations Used by Compiler

- `@Deprecated`
- `@Override`
- `@SuppressWarnings`

Why Annotation?

- Enables **declarative programming** style
 - Less coding since tool will generate the boiler plate code from annotations in the source code
 - Easier to change
- Eliminates the need for maintaining **side files** that must be kept up to date with changes in source files
 - Information is kept in the source file
 - Example- Eliminate the need of deployment descriptor

How to Define Annotation Type?

- Annotation type definitions are similar to normal Java **interface** definitions
 - An at-sign (**@**) precedes the **interface** keyword
 - Each method declaration defines an element of the annotation type
 - Method declarations must not have any parameters or a throws clause
 - Return types are restricted to primitives, String, Class, enums, annotations, and arrays of the preceding types
 - Methods can have default values

Annotation Type Definition

```
public @interface RequestForEnhancement
{
    int id();
    String synopsis();
    String engineer() default "[unassigned]";
    String date() default "[unimplemented]";
}
```

How To Use Annotation

```
import java.util.Date;
```

```
public class App1
```

```
{
```

```
@RequestForEnhancement(
```

```
    id = 2868724,
```

```
    synopsis = "Enable time-travel",
```

```
    engineer = "Mr. Peabody",
```

```
    date = "4/1/3007")
```

```
public static void travelThroughTime(Date destination) {
```

```
}
```

```
}
```


3 Different Kinds of Annotations

- Marker annotation - An annotation type with no elements

```
public @interface Preliminary { }
```

```
@Preliminary
```

```
public class TimeTravel { ... }
```

- Single value annotation - An annotation type with a single element

- The element should be named "value"

```
public @interface Copyright {  
String value();  
}
```

- **Usage – can omit the element name and equals sign (=)**

```
@Copyright("2015 Copyright to XYZ")  
public class SomeClass { ... }
```

- Normal annotation

3 Different Kinds of Annotations

- Normal annotation - We already have seen an example

```
public @interface RequestForEnhancement {  
    int id();  
    String synopsis();  
    String engineer() default "[unassigned]";  
    String date(); default "[unimplemented]";  
}
```

- Usage

```
@RequestForEnhancement(  
    id = 2868724,  
    synopsis = "Enable time-travel",  
    engineer = "Mr. Peabody",  
    date = "4/1/3007"  
)  
public static void travelThroughTime(Date destination)  
{ ... }
```

Meta-Annotation - @Retention

- How long annotation information is kept
- Enum RetentionPolicy
- **SOURCE** - SOURCE indicates information will be placed in the source file but will not be available from the class files
- **CLASS** (Default)- CLASS indicates that information will be placed in the class file, but will not be available at runtime through reflection
- **RUNTIME** - RUNTIME indicates that information will be stored in the class file and made available at runtime through reflective APIs

Meta-Annotation - @Target

- Restrictions on use of this annotation
- Enum ElementType
 - TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE, ANNOTATION_TYPE, PACKAGE

Definition and Usage of an Annotation with Meta Annotation

- Definition of Accessor annotation

@Target(ElementType.FIELD)

@Retention(RetentionPolicy.CLASS)

public @interface Accessor {

String variableName();

String variableType() default "String";

}

- Usage Example of the Accessor annotation

@Accessor(variableName = "name")

public String myVariable;

Reflection and Metadata

- **Marker annotation**

boolean isBeta =

- `MyClass.class.isAnnotationPresent(BetaVersion.class);`

- **Single value annotation**

String copyright =

`MyClass.class.getAnnotation(Copyright.class).value();`

- **Normal annotation**

String firstName

`=MyClass.class.getAnnotation(Author.class).firstName();`

String lastName

`=MyClass.class.getAnnotation(Author.class).lastName();`

Performance

Performance Basics

- **Memory footprint:** memory used by an app, or memory used by a JVM
 - Tune the heap size to avoid virtual memory swapping.
- **Startup time:** time taken to start.
 - Avoid large classpaths
 - Consider disable bytecode verification, this avoids overhead loading classes
- **Scalability:** how well an app performs under load increments
 - Lineal increments of response time is good, exponential is bad.
 - Stress your application in development, not in production
- **Responsiveness (Latency):** how quickly an app responds with a request. Respond in short period of time.
- **Throughput:** amount of work in a period of time. Do more work in the same time.

Performance methodology

■ **Monitoring**

- Not intrusive: observe behavior, not overload
- For development/production
- For troubleshooting: identify issues
- Tuning info for: JVM issues (heap size, GC optimization, JIT bugs)

■ **Profiling**

- More intrusive: analyze your code, cause overload
- For development: get performance data in a running app
- For troubleshooting: focus on the issues
- Tuning info for: code issues (memory leaks, lock contentions)

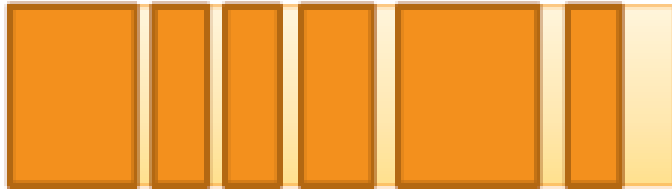
■ **Tuning**

- Known performance requirements
- Monitoring and/or profiling, then compare with requirements
- Modify JVM arguments or source code
- Incorporating performance in development
- Analysis -> Design -> Code -> Benchmark -> Profile -> Deploy

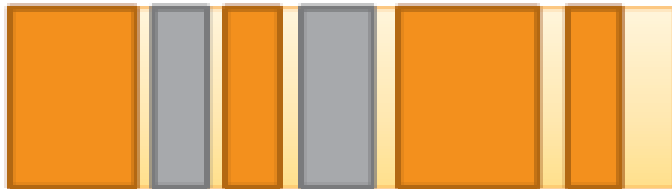
The Heap

- Each Java Application a memory area called the heap which is used for object allocation and shared by all threads running in that application
- Objects which are no longer reachable from main application tree are marked for Garbage collection
- Garbage collector is a background thread which removes the marked objects and compacts the space
- Determining default values for heap
 - `java -XX:+PrintFlagsFinal -version | grep HeapSize`
- Overriding default heap values
 - `java -Xms -Xmx`

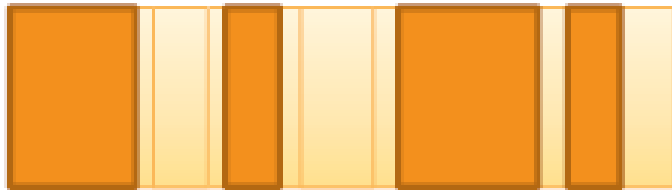
Garbage Collection on Heap



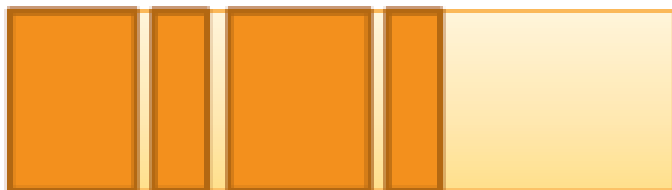
Initial State - 5 Objects allocated



Marking un-reachable - 2 Objects



Removal of marked objects



Compaction of the space

Thank You