

Santosh

**Pass Entertainment Java Coding Standard and Recommendation**

Version 1.0

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| Revision | Author | Date | Comments |
| 1.0 | Santhosh | 3/18/2018 | First draft |
|  |  |  |  |
|  |  |  |  |

Table of Contents

1 Introduction 3

1.1 Purpose 3

1.2 Scope 3

1.3 Application 3

2 Rules and Recommendations 4

2.1 General 4

2.1.1 Rules: 4

2.1.2 Recommendations: 4

2.2 Coding Style 4

2.2.1 Rules: 4

2.2.2 Recommendations: 5

2.3 Javadoc Comments 6

2.3.1 Rules: 6

2.3.2 Recommendations: 6

2.4 Files 6

2.4.1 Rules: 7

2.4.2 Recommendations: 7

2.5 Naming Conventions 7

2.5.1 Rules: 7

2.5.2 Recommendations: 8

2.6 Class Member Access 8

2.6.1 Rules: 8

2.6.2 Recommendations: 9

2.7 Class Constructors & Initialization 10

2.7.1 Rules: 10

2.7.3 Recommendations: 10

2.8 Variables and Types 10

2.8.1 Rules: 10

2.9 Flow Control Statements 11

2.9.1 Rules: 11

2.9.2 Recommendations: 13

2.10 Comments 14

2.10.1 Recommendations: 14

2.11 Exception-Handling 15

2.11.1 Rules: 15

3 Glossary 18

# Introduction

## Purpose

This document describes the Java coding standard to be adhered to when writing software at Pass Entertainment.

## Scope

This document is intended to be used by Pass Entertainment software developers.

## Application

In a team development environment, programmers often need to maintain code developed by other programmers. When a wide variety of coding styles exist in that environment, maintenance usually takes longer and is more defect-prone. Coding to a standard style creates an end product that is easier to read and maintain. Following an agreed-upon set of rules when programming also allows tools and utilities to be built that gather information about the source code.

If the rules and recommendations are followed, the result source code should be easy to maintain, have a style consistent with that produced by other developers on the project, and be free of practices that are error-prone.

The following is a list of some of the positive and negative impacts of using a set of coding standards:

|  |  |
| --- | --- |
| **Consistency**: | The team of developers can share a common view when reading, inspecting, and maintaining code. |
| **Reduce errors**: | The coding standards restrict the use of error-prone language features, and formatting. |
| **Training**: | Writing good code takes experience. If all existing code meets a minimal standard, new programmers are provided with good examples to start from. |
| **Freedom**: | Developer implementation freedom is reduced. |

# Rules and Recommendations

## General

### Rules:

1. Every time a rule in the coding standard is broken, there must be a clearly documented reason for doing so.

No coding standard can anticipate every problem that a programmer will face, and this one doesn’t pretend to. But be prepared to defend a divergence from the standard, and do so in a comment. This will prevent a well-intentioned attempt to fix the code later.

1. When there is a difference between this coding standard and that of the client, the client’s coding standard will take precedence.

### Recommendations:

1. Code for readability first. Optimize for performance only when there’s a demonstrated and proven performance problem, and a clearly understood working solution is already available.

**Exceptions:** when it's known beforehand that the performance of the code will be a critical bottleneck (e.g., network interface, disk access, etc), and where this coding standard indicates a practice for performance reasons.

## Coding Style

### Rules:

1. One level of indentation equals four spaces. **Tabs must be replaced with four spaces[[1]](#footnote-1).**
2. Line length should not exceed 120 characters.
3. Opening and closing braces must be formatted using one of the following two methods:
4. On a line by themselves at the same level of indentation as the initiating keyword.

if ( byteBuf == null )  
{  
 byteBuf = new ByteBuffer( BUFFER\_SIZE );  
}

**Exception:** the closing brace of the do whle statement.

do  
{  
 charNumber++;  
 ch = clientName.getChar( charNumber );  
  
} while ( ch != '\0' );

1. Opening brace on the same line as the initiating keyword and closing brace on a line by itself at the same level of indentation as the initiating keyword.

if ( byteBuf == null ) {  
 byteBuf = new ByteBuffer( BUFFER\_SIZE );  
}

**Exception:** the closing brace of the do while and try statements.

do {  
 charNumber++;  
 ch = clientName.getChar( charNumber );  
   
} while ( ch != '\0' );  
   
  
try {  
 doSomeThing()  
} catch (ExceptionClass e) {  
 handleException();  
}

The two methods for formatting braces must not be mixed in same file or package. When modifying an existing file or package, the existing brace formatting style must be used and not changed[[2]](#footnote-2).

### Recommendations:

1. Always write the left parenthesis directly after a function name (no intervening space).
2. Surround operators with spaces.

Don't do things like this:

newXPos+=recClient.Width()+recUpdate.Width()+margin\*columns;

It makes the operators easy to miss in a quick inspections. Do this instead:

newXPos += recClient.Width() + recUpdate.Width() + margin \* columns;

1. When a statement spans more than one line, break the statement in a way that each line is obviously syntactically incorrect, and indent the continued line or lines in a way that makes the continuation more visible..

This prevents misreading of code. For example, the first line of the statement below could be interpreted as a complete statement:

String addressLine1 = getApartmentNumber() + getStreetAddress() + getStreetName();

The code below provides a strong visual cue that it’s continued on the next line.

String addressLine1 = getApartmentNumber() + getStreetAddress()   
 + getStreetName();

## Javadoc Comments

### Rules:

1. All classes and public methods must be preceded by a javadoc comment.
2. Class javadoc comments must contain a meaningful description of the class and an @author tag for each person who has worked on an part of the class.

/\*\*  
<p> COPYRIGHT </p>  
<p> Pass Entertainment -- Copyright (C) 2018 Pass Entertainment  
 All rights reserved. No part of this computer program  
 may be used or reproduced in any form by any  
 means without prior written permission of  
 Pass Entertainment. </p>

<p>DESCRIPTION </p>  
<p>Contains the dimensions of a bounding box that will enclose  
zero or more graphical objects on a screen. The dimensions  
will increase as required when items are added to the box. </p>  
  
@author Santhosh  
@version 1.0  
@see related\_class  
\*/  
class BoundingBox  
{ ...

1. Method javadoc comments must include a meaningful description of the method @param tags for each parameter, and, if applicable, @return and @exception tags.

/\*\*  
Adds a graphic object to the bounding box. If the any of the object’s  
dimensions fall outside the box, the box’s dimensions will grow to  
include them.  
  
@parameter element the graphic element to be added to the box  
  
@return true if the bounding box has expanded as a result of adding  
 the item, false if the dimensions are unchanged  
@see see\_related\_method or material  
\*/  
public boolean addItem( GraphicElement element )  
{ ...

### Recommendations:

1. Precede all non-public methods by a javadoc comment.

## Files

### Rules:

All files or packages must include the standard comment header with copyright information as required by the project.-1999<**company name**><**company name**>

### Recommendations:

1. Try to keep the size of source files to less than one thousand lines.

If source files become larger than this, there's a good chance that the design should be looked at again, with more functionality being moved into smaller objects.

## Naming Conventions

### Rules:

1. Whenever mixed-case names are made up of more than one word, the first letter of each word following the first should be uppercase.

closeFile();  
Button okButton;  
int accountTotal;  
int average;

1. Whenever all-uppercase names are made up of more than one word, each word should be separated with an underscore.

final int MAX\_RANGE = 100;  
final String BUTTON\_NAME = “a button name”;

1. Methods should be named so that they describe what the method does, starting with an active verb whenever possible.

protected void hideComponent() ...  
public int getAverage() ...  
public void refreshScreen() ...

1. Accessor methods that retrieve an attribute should begin with “get”.
2. Modifier methods should begin with “set”.
3. Components should end with the type of the component.

This is to make the use of the variable a bit more obvious. For example “ageTextField” is unambiguous, while “age” may be mistaken for a numeric value.

Button cancelButton = new Button();  
List customerNameList;  
BigGrid salesTrackingGrid;

1. Class names should be a description of the class, and should be mixed case and begin with an uppercase letter.
2. Interface names should be a description of the interface’s services.

interface Compressable ...  
interface Singleton ...

1. Package names should be all lower case, and be prefixed with the Internet domain of your organization, but with the components of the domain name in reverse order. For example, a for-profit organization with a domain "ourcomany.com" should create packages with names like "com.ourcompany.ourproject", "com.ourcompany.anotherproject", and "com.ourcompany.ourproject.packageabc". Do not use capitals for the class A prefix.
2. File names should be camel case. Java source files should match the name of the class they contain. Whether the first letter of the filename is upper or lower case should depend on the use of the file, standard conversions should be followed. Filenames should only contain ‘a’-‘z’, ‘A’-‘Z’, ‘0’-‘9’ and ‘\_’.

### Recommendations:

1. Avoid the overuse of “i”, “j”, and “k” as variable names.

These variable names don’t offer much in the way of providing readability. Compare:

for ( int i = 0; i < totalMonthsDisplayed; i++ )  
{  
 for ( j = 0; j < totalSales; j++ )  
 {  
 refresh( i, j );  
 }  
}

... with ...

for ( int row = 0; row < totalMonthsDisplayed; row++ )  
{  
 for ( column = 0; column < totalSales; column++ )  
 {  
 refresh( row, column );  
 }  
}

## Class Member Access

### Rules:

1. All non-final fields should be private.

Hiding data provides all the benefits of using abstract data types, and limits the places that data can be changed (and/or corrupted). Giving an object the ability to prevent access to its internal state also allows it to test itself whenever its state is queried or changed to ensure that the change is consistent with the object’s internal state.

Don't do this:

public class Foo  
{  
 public int counter;  
}

Do this instead:

public class Foo  
{  
 private int counter;  
  
 public int getCounter( void )  
 {  
 return counter;  
 }  
 public void setCounter( int newCounter )  
 {  
 counter = newCounter;  
 }  
}

Another problem with public, protected, or package access fields becomes apparent when a variable name must be changed or removed from the class. Every derived class (or friends of the derived classes) that directly reference those variables must also be modified.

2. Avoid having public member function return a reference to a private field.

This allows outside code to change the state of an object without that object knowing about it. When possible, return a clone of the member object.

### Recommendations:

1. When a private data member has a public or protected access method, those methods should be used internally by the class as much as possible to provide consistency.

For example, given the following class:

class Dimensions  
{  
 public void setDimensions( int width, int height )  
 { ... }  
   
 public int getWidth( void ) { ... }  
 public int getHeight( void ) { ... }  
 public int getArea( void ) { ... }  
  
 private int width; **// width dimension**  
 private int m\_height; **// height dimension**  
}

The code in the body of Dimension.getArea() could look like this:

area = width \* height:

This would work, but will need to be modified if the height or width were changed from a data member to a calculated value. The following code would require no such modification:

area = getWidth() \* getHeight();

## Class Constructors & Initialization

### Rules:

1. Instance initialization code blocks should only be used in anonymous classes.

Putting construction code in constructors keeps all this related code together. When various code blocks initialize data throughout the class definition, understanding what’s being initialized by whom becomes difficult.

1. Always explicitly call the base class constructor in any place that the compiler would implicitly call it.

In any constructor that does not have as its first line of code (a) a call to a base class constructor, or (b) a call to another “this” constructor, the compiler will implicitly insert a call to super(). This can lead to confusion when the base class has no no-argument constructor.

An additional benefit is that tracing execution through a constructor chain makes more sense when there are explicit calls to super().

1. Always explicitly create any constructor that the compiler would implicitly create, even if it contains no statements other than a call to super().

The compiler will explicitly create an empty no-argument constructor when you define no constructors for a class. Explicitly coding one prevents some confusion when debugging, and makes more sense when tracing through constructor chains.

### Methods

### Recommendations:

1. Avoid methods with many arguments.
2. Avoid long and complex methods.

## Variables and Types

### Rules:

1. Avoid the use of numeric values in code, use symbolic values instead.

For example:

colWidth = strWidth + 10 + ( level \* 3 );  
...  
cellWidth = cellStrWidth + 10;

The unfortunate programmer who has to maintain (or fix) the above code, there's no indication of what “10” or “3” represents, or whether both “10”s represent the same thing. And if the code maintainer finally figures out that “10” probably represents margin widths, and changes it to something smaller, there's an excellent chance that an instance will be missed, and more time will be wasted tracking down a difficult-to-find bug. Something like the following code should be written in the first place:

final int COL\_INDENT\_WIDTH = 3; **// amount to indent, per level**  
final int COL\_LEFT\_MARGIN = 5; **// width of column left margin in pixels**  
final int COL\_RIGHT\_MARGIN = 5; **// width of column right margin in pixels**  
  
**// pixel width of both margins combined**final int COL\_MARGINS\_WIDTH = COL\_LEFT\_MARGIN + COL\_RIGHT\_MARGIN;  
  
colWidth = strWidth + COL\_MARGINS\_WIDTH + ( level \* COL\_INDENT\_WIDTH );  
...  
cellWidth = cellStrWidth + COL\_MARGINS\_WIDTH;

1. Variables are to be declared with the smallest possible scope.
2. Each variable is to be declared in a separate declaration statement, and have a commented description.

Don't do this:

int moveAmount, selectedIndex;

Do this:

int moveAmount; **// number of items to move**  
int selectedIndex; **// currently selected item**

## Flow Control Statements

### Rules:

1. The code which following a **case** label must always be terminated by a break statement.

The switch statement must always contain a default branch, which handles unexpected case.

switch (tag)

{

case A:

{

// Do something

break;

}

default:

{

// If no match in above cases, this is executed

}

}

**Exception:** when several different cases have identical handling.

1. Use opening and closing braces in case blocks that contain variable declarations.
2. A switch statement must always contain a default branch which handles unexpected cases.
3. All flow control statements (if, else, while, for and do) must have opening and closing braces, even if the block contains no statements.

Empty loops can be deceiving. In the example below, it's easy to mistakenly think that the last line is contained in the loop:

for ( int rowNumber = initialRow;  
 ( rowNumber < totalRows ) && !getRow( rowNumber ).getIsSelected() &&  
 ( getRow( rowNumber ).getNumberOfCells() > 0 );  
 rowNum++ );  
numberOfRowFound = rowNumber;

When the braces are included, the empty loop is unmistakable:

for ( int rowNumber = initialRow;  
 ( rowNumber < totalRows ) && !getRow( rowNumber ).getIsSelected() &&  
 ( getRow( rowNumber ).getNumberOfCells() > 0 );  
 rowNumber++ )  
{  
 // empty  
}  
numberOfRowFound = rowNumber;

1. Use an explicit if...else... structure rather than the ternary statement.

The ternary statement acts as a shortcut for the if...else... statement. However, the statement:

( rowNumber == 0 ) ? selectTable() : selectRow( rowNumber );

...will likely compile the same as the more readable:

if ( rowNumber == 0 )  
{  
 selectTable();  
}  
else  
{  
 selectRow( rowNumber );  
}

**Exception**: cases in which a ternary statement simplifies an assignment.

int double scale = ( isHalfScale() ? 0.5 : 1 );  
  
**// ...may be more readable than...**  
int double scale;  
if ( isHalfScale() )  
{  
 scale = 0.5;  
}  
else  
{  
 scale = 1;  
}

### Recommendations:

1. Use inclusive lower limits and exclusive upper limits.

Rather than say that x is the interval x >= 0 and x <= 9, use the limits x >= 0 and x < 10. If you do so, then:

1. the difference in the limits is the size of the interval (interval = 10-0 rather than interval = 9‑0+1)
2. if the interval is zero, the limits will be equal
3. the upper limit will never be less than the lower limit.
4. Avoid the use of continue.

continue tends to make loops harder to understand by concealing the structure of program execution. When tempted to use them, try to use an if...else.. statement instead.

**Exception:** a continue statement near the top of a loop can sometimes make the code more readable by avoiding a series of if...else... statements, or a complex loop control structure. For example:

while ( row != null )  
{  
 if ( row.getIsEmpty() )  
 {  
 **// row cannot contain data, do not process** continue;  
 }  
 ... process row ...  
 row = row->getNext();  
}

1. Avoid the use of break in anything but the switch statement.

break tends to make loops harder to understand by concealing their exit conditions. Here's an example of break that confuses the structure of a loop:

while ( true )  
{  
 Rectangle areaRect = area->GetRectArea();  
 if ( areaRect.pointInRect( firstPoint ) )  
 {  
 if ( area.getType() == NON\_CLIENT\_AREA )  
 areaWidth -= areaRect.Width()  
 else  
 break;  
 }  
 if ( area->getNext() == null )  
 break;  
 else  
 area = area->Next();  
}

**Exception:** break can sometimes be used judiciously to avoid messy nested if...else statements in loops or a large number of boolean conditions in the control statement at the top:

while ( ch != '\u0000' )  
{  
 **// if this is a metastring token, drop out of the loop**  
 if ( ch == '%' )  
 {  
 tokenPosition = pos;  
 break; **// <------ early exit from loop** }  
 ... more code to process 'ch' ...  
}

1. Avoid the use of multiple return statements.

The flow of control within a function is more difficult to understand when there are multiple exit points.

**Exception:** early returns can sometimes be used at the beginning of a function to avoid long if...else.. constructions. When used in this way, it's recommended that they are documented in a way that draws attention to them.

boolean setName( Client client, String name )  
{  
 int clientIndex = findClient( client );  
 if ( clientIndex == CLIENT\_NOT\_FOUND )  
 {  
 return; **// <--------- early return for invalid client**  
 }  
 ...

## Comments

### Recommendations:

1. Comment in a style that's easy to maintain.

Example of a hard-to-maintain commenting style:

//--------------------------------------//  
// get the size of the buffer //  
//-------------------------------------//

Example of easy-to-maintain commenting styles:

//------------------------------  
// get the size of the buffer  
//------------------------------  
  
/\*  
 \* get the size of the buffer  
 \* then allocate another buffer of the same size.  
 \*/

1. Comment as you go.

Don't code with the intention of going back later and putting comments in. Try to comment first and code around the comments.

1. Avoid non-data endline comments.

Endline comments are difficult to maintain because they have to aligned properly, are difficult to edit, and there usually isn't enough space to comment effectively.

Exceptions: data declarations, and comments at the end of blocks. Example of these:

int numberOfKeys; **// number of virtual keystrokes**...  
 } **// switch**   
} **// for**

1. Don't duplicate the code in the comment.

The following comments are redundant:

**// if the allocation flag is more than zero**if ( allocSize > 0 )  
{  
 **// initialize buffer size to allocSize** bufferSize = allocSize;  
}

This comment explains what's actually happening:

**// if a buffer was successfully allocated, record the size**if ( allocSize > 0 )  
{  
 bufferSize = allocSize;  
}

## Exception-Handling

### Rules:

1. Where there is a clear separation between presentation classes and integration classes, the integration class should repackage checked exceptions to pass to the presentation class to preserve encapsulation.

public class UserDAOImpl implements UserDAO  
{  
 findByUserId(String userId)  
 {  
 User user = null;  
 try   
 {  
 user = (User)getSqlMapClientTemplate().queryForObject(GET\_USER, userId);  
 }  
 catch(DataAccessException ex)  
 {  
 //service class, must repackage exceptionlogger.error(“could not access database”,ex);  
 **throw new applicationException(“could not connect to database”, ex);**  
 }  
 }

1. Preserve the original stack trace information throughout the life of the Exception.

If an exception is caught in an integration class, and an exception is to be re-thrown to the presentation class, the newly thrown exception must preserve the caught exception stack trace

1. Exceptions should be logged once whenever they are caught except as outlined by rule 4 and dealt with where they are caught, exceptions should not be passed to generic exception handling methods.

public class UserDAOImpl implements UserDAO  
{  
 findByUserId(String userId)  
 {  
 User user = null;  
 try   
 {  
 user = (User)getSqlMapClientTemplate().queryForObject(GET\_USER, userId);  
 }  
 catch(DataAccessException ex)  
 {  
 **//service class, must repackage exception  
 logger.error(“could not access database”,ex);**  
 throw new ApplicationException(“could not connect to database”, ex);  
 }  
 }

1. Swallowed checked exceptions should be avoided, and where they cannot be avoided proper commenting should document why an empty catch block exists. The general exception java.lang.Exception should never be caught.

try   
{  
 AdditionalData additionalData = getAdditionalUserData(userId);  
}  
catch(UserNotRegistered ex)  
{  
 **/\*  
 \* thrown if the user has not been registered in the third party service  
 \* which isn’t necessary for updating the user information  
 \*/**   
}

1. A function should not contain split try catch blocks, where multiple checked exceptions are caught they should be handled in subsequent catch blocks or if necessary in nested try blocks.

try   
{  
 User user = (User)getSqlMapClientTemplate().queryForObject(GET\_USER, userId);  
 UserMetaData userMetaData = userMetaDataService.getUserMetaDataByUser(user);  
}  
catch(SQLException ex)  
{  
 **throw new ApplicationException(“could not access database” , ex);**  
}  
catch(WebServiceConnectionException ex)  
{  
 **throw new ApplicationException(“could not connect to web service” , ex);**  
}

1. List all normal exceptions thrown in the throws clause, not just base class of related exceptions.

In other words, if your method throws MyException and MyDerivedException (which is derived from MyException), don’t just declare MyException. While this is legal in the language, it hides useful information from those using your class.

1. Catch subclasses before base classes.

If you catch the base class of an exception before trying to catch classes derived from it, the first catch block will catch all derived exceptions:

try  
{  
 doSomething();  
}  
catch ( BaseExceptionClass e ) **// this should come after the next block**  
{  
 ...  
}  
catch ( DerivedFromBaseExceptionClass e )  
{  
 ... **// will never get to here, because the previous catch  
 // statement will catch all classes derived from   
 // BaseExceptionClass**    
}

# Glossary

abstract class

A class that exists only as a superclass of another class and can never be directly instantiated. In Java, an abstract class contains or inherits one or more abstract methods or includes the abstract keyword in its definition.

abstract method

A method that has no implementation.

abstract data type

Defines a type that may have many implementations. Abstract data types include things like stacks, queues, and trees.

abstract type

Defines the type for a set of objects, where each object must also belong to a set of objects that conform to a known subtype of the abstract type. An abstract type may have one or more implementations.

abstraction

The process and result of extracting the common or general characteristics from a set of similar entities.

accessor

A method that sets or gets the value of an object property or attribute.

active object

An object that possesses its own thread of control.

acyclic dependency

A dependency relationship where one entity has a direct or indirect dependency on a second entity, but the second entity has no direct or indirect dependency on the first.

aggregation

An association representing a whole-part containment relationship.

architecture

A description of the organization and structure of a software system.

argument

Data item specified as a parameter in a method call.

assertion

A statement about the truth of a logical expression.

attribute

A named characteristic or property of a type, class, or object.

behavior

The activities and effects produced by an object in response to an event.

binary compatible

A situation where one version of a software component may be directly and transparently substituted for another version of that component without recompiling the component's clients.

block statement

The Java language construct that combines one or more statement expressions into a single compound statement, by enclosing them in curly braces "{...}".

###### Boolean

An enumerated type whose values are true and false.

built-in type

A data type defined as part of the language. The built-in or native types defined by Java include the primitive types boolean, byte, char, double, float, int, long, short, and void, and the various classes and interfaces defined in the standard Java API, such as Object, String, Thread, and so forth.

checked exception

Any exception that is not derived from java.lang.RuntimeException or java.lang.Error, or that appears in the throws clause of a method. A method that throws, or is a recipient of, a checked exception must handle the exception internally or otherwise declare the exception in its own throws clause.

class

A set of objects that share the same attributes and behavior.

class hierarchy

A set of classes associated by inheritance relationships.

client

An entity that requests a service from another entity.

cohesion

The degree to which two or more entities belong together or relate to each other.

component

A physical and discrete software entity that conforms to a set of interfaces.

composition

A form of aggregation where an object is composed of other objects.

concrete class

A completely specified class that may be directly instantiated. A concrete class defines a specific implementation for an abstract class or type.

concrete type

A type that may be directly instantiated. A concrete type may refine or extend an abstract type.

concurrency

The degree by which two or more activities occur or make progress at the same time.

constraint

A restriction on the value or behavior of an entity.

constructor

A special method that initializes a new instance of a class.

container

An object whose purpose is to contain and manipulate other objects.

contract

A clear description of the responsibilities and constraints that apply between a client and a type, class, or method.

coupling

The degree to which two or more entities are dependent on each other.

critical section

A block of code that allows only one thread at a time to enter and execute the instructions within that block. Any threads attempting to enter a critical section while another thread is already executing within that section are blocked until the original thread exits.

cyclic dependency

A dependency relationship where one entity has a direct or indirect dependency on a second entity and the second entity also has a direct or indirect dependency on the first.

data type

A primitive or built-in type that represents pure data and has no distinct identity as an object.

delegation

The act of passing a message, and responsibility, from one object to a second object to elicit a desired response.

dependency

A relationship where the semantic characteristics of one entity rely upon and constrain the semantic characteristics of another entity.

derivation

The process of defining a new type or class by specializing or extending the behavior and attributes of an existing type or class.

generalization

The process of extracting the common or general characteristics from a set of similar entities to create a new entity that possesses these common characteristics.

documentation comment

A comment that begins with a "/\*\*" and ends with “\*/”and contains a description and special tags that are parsed by the Javadoc utility to produce documentation.

domain

An area of expertise, knowledge, or activity.

encapsulation

The degree to which an appropriate mechanism is used to hide the internal data, structure, and implementation of an object or other entity.

enumeration

A type that defines a list of named values that make up the allowable range for values of that type.

factor

The act of reorganizing one or more types or classes by extracting responsibilities from existing classes and synthesizing new classes to handle these responsibilities.

field

An instance variable or data member of an object.

fundamental data type

A type that typically requires only one implementation and is commonly used to construct other, more useful types. Dates, complex numbers, linked-lists, and vectors are examples of common fundamental data types.

implementation

The concrete realization of a contract defined by a type, abstract class, or interface. The actual code.

implementation class

A concrete class that provides an implementation for a type, abstract class, or interface.

implementation inheritance

The action or mechanism by which a subclass inherits the implementation and interface from one or more parent classes.

inheritance

The mechanism by which more specialized entities acquire or incorporate the responsibilities or implementation of more generalized entities.

###### inner class

A class defined within the scope of another class.

instance

The result of instantiating a class-the concrete representation of an object.

instantiation

The action or mechanism by which a type or class is reified to create an actual object. The act of allocating and initializing an object from a class.

interface

The methods exposed by a type, class, or object. Also a set of operations that define an abstract service.

interface inheritance

The action or mechanism by which a subtype or subinterface inherits the interface from one or more parent types or interfaces.

invariant

An expression that describes the well-defined, legal states of an object.

keyword

A language construct. The keywords of the Java language include:

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

Bracketed keywords are reserved but not used.

lazy initialization

When an implementation delays the initialization of a data value until the first use or access of the data value.

local variable

A variable that is automatically allocated and initialized on the call "stack." Includes variables bound as function arguments.

method

The implementation of an operation. An operation defined by an interface or class.

multiple inheritance

Inheritance relationship where a subtype inherits from two or more supertypes. Java supports multiple inheritance by allowing an interface to extend multiple interfaces.

mutex

A synchronization mechanism used to provide mutually exclusive access to a resource.

native type

A data type defined as part of the language. The native or built-in types defined by Java include the primitive types boolean, byte, char, double, float, int, long, short, and void, and the various classes and interfaces defined in the standard Java API, such as Object, String, and Thread.

object

The result of instantiating a class. An entity with state, behavior, and identity.

operation

A service that may be requested from an object to effect behavior. Alternatively viewed as a message sent from a client to an object.

package

A mechanism organizing and naming a collection of related classes.

package access

The default access-control characteristic applied to interfaces, classes, and class members. Class members with package access are accessible only to code within the same package and are heritable by subclasses in the same package. Classes and interfaces with package access are not visible to code outside the package. They are only accessible and extendable by classes and interfaces in the same package.

parameter

A variable that is bound to an argument value passed into a method.

polymorphic

A trait or characteristic of an object whereby that object can appear as several different types at the same time.

polymorphism

The concept or mechanism by which objects of different types inherit the responsibility for implementing the same operation, but respond differently to the invocation of that operation.

postcondition

A constraint or assertion that must hold true following the completion of an operation.

precondition

A constraint or assertion that must hold true at the start of an operation.

primitive type

A basic language type that represents a pure value and has no distinct identity as an object. The primitives provided by Java include boolean, byte, char, double, float, int, long, and short.

private access

An access-control characteristic applied to class members. Class members declared with the private access modifier are only accessible to code in the same class and are not inherited by subclasses.

property

A named characteristic or attribute of a type, class, or object.

protected access

An access-control characteristic applied to class members. Class members declared with the protected access modifier are accessible to code in the same class and package, and from code in subclasses, and they are inherited by subclasses.

public access

An access-control characteristic applied to interfaces, classes, and class members. Class members declared with the public access modifier are accessible anywhere the class is accessible and are inherited by subclasses. Classes and interfaces declared with the public access modifier are visible, accessible, and heritable outside of a package.

qualifier

A name or value used to locate or identify a particular entity within a set of similar entities.

realization

A relationship where one entity agrees to abide by or to carry out the contract specified by another entity.

responsibility

A purpose or obligation assigned to a type.

role

The set of responsibilities associated with an entity that participates in a specific relationship. A Java interface often defines a role for an object.

service

One or more operations provided by a type, class, or object to accomplish useful work on behalf of one or more clients.

signature

The name, parameter types, return type, and possible exceptions associated with an operation.

state

The condition or value of an object between events.

static type checking

Compile-time verification of the assumptions made about the use of object reference and data value types.

subclass

A class that inherits attributes and methods from another class.

subtype

The more specific type in a specialization-generalization relationship.

superclass

A class whose attributes and methods are inherited by another class.

supertype

The more general type in a specialization-generalization relationship.

synchronization

The process or mechanism used to preserve the invariant states of a program or object in the presence of multiple threads.

synchronized

A characteristic of a method or a block of code. A synchronized method or block allows only one thread at a time to execute within the critical section defined by that method or block.

thread

A single flow of control flow within a process that executes a sequence of instructions in an independent execution context.

type

Defines the common responsibilities, behavior, and operations associated with a set of similar objects. A type does not define an implementation.

unchecked exception

Any exception that is derived from java.lang.RuntimeException or java.lang.Error. A method that throws, or is a recipient of, an unchecked exception is not required to handle the exception or declare the exception in its throws clause.

variable

A typed, named container for holding object references or a data values.

visibility

The degree to which an entity may be accessed from outside of a particular scope.

1. Replacing tabs with spaces is done so that code will be formatted properly when viewed on UNIX systems, which use a tab spacing of 8. [↑](#footnote-ref-1)
2. Although some development environments provide automated capabilities to easily convert between the two brace formatting styles, doing will disrupt version control or configuration management systems which track changes between different versions of a file (making it difficult to identify changes between file versions). [↑](#footnote-ref-2)