SUN CERTIFIED PROGRAMMER FOR JAVA 2

This is a complete set of notes containing every thing you need to know for the SUN CERTIFIED PROGRAMMER FOR JAVA 2 EXAMS 310-035 & 310-027

Source Files

- If a source code file does not contain a public class or interface it can take on a name that is different from its classes and interfaces.
- A source code file cannot contain more than one public class or interface.

Imports

- The *java.lang* package is always imported by default and does not need to be imported by an import statement.

Java Programming Language Keywords

- Keywords cannot be used as identifiers (names) for classes, methods, variables, or anything else in your code.
- All keywords start with a lowercase letter.

Literals and Ranges of All Primitive Data Types

- All six number types in Java are signed, so they can be positive or negative.
- Use the formula $-2^{(bits-1)}$ to $2^{(bits-1)}-1$ to determine the range of an integer type.
- A char is really a 16-bit unsigned integer (Unicode character).
- Literals are source code representations of primitive data types, or a *String*.
- Integers can be represented in octal (0127), decimal (1245), and hexadecimal (0XCAFE or 0xCAFE).
- Suffixes:
    - double ***d*** or ***D*** (64-bit signed floating-point number)
    - long ***l*** or ***L*** (64-bit signed integer)
    - float ***f*** or ***F*** (32-bit signed floating-point number)
- Floating-point numbers are assumed to be *double* if no suffix is specified.
- Legal formats:

    digits . optionalDigits optionalExponent suffix

    . digits optionalExponent suffix

digits optionalExponent suffix

- The optional exponent part consists of an *e* or *E* followed by a signed integer.
- If a floating-point literal does not contain a decimal point then it needs to have either the exponent part or the suffix to be recognised as a floating-point literal as opposed to an integer literal.
- Character escape codes
    - ***b*** – backspace
    - ***t*** – tab
    - ***n*** – line-feed
    - ***f*** – form-feed
    - ***r*** – carriage return
    - **"** – double quote
    - **'** – single quote
    - \\ – backslash
- The backslash can also be followed by an 8-bit octal value *\000* through *\377* or by *u* or *U* followed by a four-digit hexadecimal value *\u0000* (or *\U0000*) through *\uffff* (or *\UFFFF*). The four digit hexadecimal value can be used to specify the full range of Unicode characters.
- A class literal is formed by appending *.class* to the name of a primitive or reference type. It evaluates to the class descriptor of the reference type or class descriptor of the primitive type's wrapper class.
- Default values:
    - *boolean* *false*
    - *byte* *0*
    - *char* *\u0000*
    - *short* *0*
    - *int* *0*
    - *long* *0l*
    - *float* *0.0f*
    - *double* *0.0*

Array Declaration, Construction, and Initialization

- Arrays can hold primitives or objects, but the array itself is always an object.
- When you declare an array, the brackets can be to the left or right of the variable name.

- It is never legal to include the size of an array in the declaration.
- You must include the size of an array when you construct it using *new* unless you are creating an anonymous array. i.e:

  String[] weeks = new String[52];
  String[] days = new String[] {"Su", "Mo", "Tu", "We", "Th", "Fr", "Sa"};

- You can also create and initialize arrays without the *new* keyword although this can only be done in an array declaration i.e:

  String[] days = {"Su", "Mo", "Tu", "We", "Th", "Fr", "Sa"};

- Elements in an array of objects are not automatically created, although primitive array elements are given default values.
- You'll get a *NullPointerException* if you try to use an array element in an object array, if that element does not refer to a real object.
- Arrays are indexed beginning with zero. In an array with three elements, you can access element 0, element 1, and element 2.
- You'll get an *ArrayIndexOutOfBoundsException* if you try to access outside the range of an array.
- Arrays have a length variable that contains the number of elements in the array.
- The last index you can access is always one less that the length of the array.
- Multidimensional arrays are just arrays of arrays.
- The dimensions in a multidimensional array can have different lengths.
- An array of primitives can accept any value that can be promoted implicitly to the declared type of the array. For example, a *byte* or *short* can be placed in an *int* array.
- An array of objects can hold any objects that passes the IS-A (or *instanceof*) test for the declared type of the array. For example, if *Horse* extends *Animal* then a *Horse* object can go into an *Animal* array.
- You can assign an array of one type to a previously declared array reference of its supertypes. For example, a *Honda* array can be assigned to an array declared as type *Car* (assuming *Honda* extends *Car*).

Using a Variable or Array Element that is Uninitialized and Unassigned

- When an array of objects in instantiated, objects within the array are not instantiated automatically, but all the references get the default value of *null*.
- When an array of primitives is instantiated, all elements get their default values.
- Just as with array elements, instance variables are always initialized with a default value.
- Local / automatic / method variables are never given a default value. If you attempt to use one before initializing it you'll get a compiler error.

Class Access Modifiers

- There are three access modifiers: *public, protected* and *private.*
- There are four access levels: *public, protected*, default and *private*
- Classes can have only *public* or default access
- Class visibility revolves around whether code in one class can:
    - Create an instance of another class
    - Extend (or subclass), another class
    - Access methods and variables of another class
- A class with default access can be seen only by all classes within the same package.
- A class with public access can be seen by all classes from all packages.

Class Modifiers (non-access)

- Classes can also be modified with *final, abstract* or *strictfp.*
- A *class* that is *abstract* cannot also be *final.*
- A method that is *abstract* cannot also be *final* or *private.*
- A *final* class cannot be subclassed.
- An *abstract* class cannot be instantiated.
- A single *abstract* method in a class means the whole class must *abstract* and must be declared as so.
- An *abstract* class can have both abstract and non-abstract methods.
- The first concrete class to extend an *abstract* class must implement all *abstract* methods.

Member Access Modifiers

- Methods and instance (nonlocal) variables are known as "members".
- Members can use all four levels: *public*, *protected*, default and *private.*

- Member access comes in two forms:
    - Code in one class can access a member of another class.
    - A subclass can inherit a member of its superclass.
- If a *class* cannot be accessed, its members cannot be accessed.
- Determine *class* visibility before determining member visibility.
- *public* members can be accessed by all other classes, even in different packages.
- If a superclass member is *public*, the subclass inherits it – regardless of *package*.
- *this.* Always refers to the currently executing object.
- *private* members can be accessed only by code in the same *class*
- *private* members are not visible to subclasses, so *private* members cannot be inherited, and therefore cannot be *abstract*.
- Default and *protected* members differ only in when subclasses are involved:
    - Default members can be accessed only by other classes in the same *package*.
    - *protected* members can be accessed by other classes in the same *package*, plus subclasses regardless of the *package*.
- *protected* = *package* + subclasses
- Default = *package* only
- A *protected* member inherited by a subclass from another *package* is not accessible to any other *class* in the subclass *package*, except for the its own subclasses.

Local Variables

- Local (method, automatic, stack) variable declarations cannot have access modifiers.
- *final* is the only modifier available to local variable.
- Local variables don't get default values, so they must be initialized before use.

Other Modifiers – Members

- *final* methods cannot be overridden in a subclass
- *abstract* methods must be declared, with a signature and return type, but are not implemented.
- *abstract* methods end in a semicolon – no curly braces.
- Three ways to spot a nonabstract method:
    - The method is not marked abstract
    - The method has curly braces

    - The method has code between the curly braces
- The first nonabstract (concrete) *class* to extend an *abstract class* must implement all of the *abstract class*' *abstract* methods.
- *abstract* methods must be implemented by a subclass, so they must be inheritable. For that reason:
    - *abstract* methods **cannot be** *private.*
    - *abstract* methods **cannot be** *final.*
    - *abstract* methods **cannot be** *synchronized.*
    - *abstract* methods **cannot be** *native.*
    - *abstract* methods **cannot be** *strictfp.*
- *abstract* class' **can be** *strictfp.*
- The strictfp modifier is used in front of a method or class to indicate that floating-point numbers will strictly follow the Java Specification for floating point calculation (i.e. only 64-bit) and no more accurate. *strictfp* therefore provides repeatability for different platforms.
- The *synchronized* modifier applies only to methods.
- *synchronized* methods **can** have any access control and **can** be marked *final*.
- *synchronized* methods **cannot** be *abstract*.
- The *native* modifier applies only to methods.
- The *strictfp* modifier applies only to classes and methods.
- Instance variables can
    - Have any access control
    - Be marked *final* or *transient*
- Instance variables cannot be declared *abstract*, *synchronized, native* or *strictfp.*
- It is legal to declare a local variable with the same name as an instance variable; this is called "shadowing"
- *final* variables have the following properties
    - *final* variables cannot be reinitialized once assigned a value.
    - *final* reference variables cannot refer to a different object once the object has been assigned to the *final* variable.
    - *final* reference variables must be initialized before the constructor completes.
- There is no such thing as a *final* object. An object reference marked *final* does not mean the object itself is immutable.

- The *transient* modifier applies only to instance variables.
- The *volatile* modifier applies only to instance variables.
- The *transient* modifier prevents field from being serialized, as transient fields are always skipped when objects are serialized.
- The *volatile* modifier is used on variables that may be modified simultaneously by other threads. This warns the compiler to fetch them fresh each time, rather than caching them in registers. This also inhibits certain optimisations that assume no other Thread will change the values unexpectedly. Since other threads cannot see local variables, there is never any need to mark local variables *volatile*.

Static variables and methods

- They are not tied to any particular instance of a class
- An instance of a *class* does not need to exist in order to use *static* members of the *class*.
- There is only one copy of a *static* variable per *class* and all instances share it.
- *static* variables get the same default values as instance variables.
- A *static* method (such as *main( )*) cannot access a nonstatic (instance) variables.
- *static* members are accessed using the class name as follows *ClassName.theStaticMethod( ).*
- *static* members can also be accessed using an instance reference variable, *someObj.theStaticMethod( )* but that's just a syntax trick. The complier uses the *class* type of the reference variable to determine which *static* method to invoke.
- *static* methods cannot be overridden, although they can be redeclared / redefined by a subclass. So although *static* methods can sometimes appear to be overridden, polymorphism will not apply.

Declaration Rules

- A source code file can have only one *public class*.
- If the source file contains a *public class*, the file name should match the *public class* name.
- A file can have more than one non-public *class*.
- Files with no *public* classes have no naming restrictions.
- In a file, classes can be listed in any order (there is no forward referencing problem).
- *import* statements only provide a typing shortcut to a class' fully qualified name.

- *import* statements cause no performance hits and do not increase the size of your code.
- *imports* ending in .*; are importing all classes within a *package*.
- *imports* ending in ; are importing a single *class*.
- You must use fully qualified names when you have different classes from different packages, with the same *class* name; an *import* statement will not be explicit enough.

Properties of main( )

- It must be marked *static*.
- It must have a *void* return type.
- It must have a single *String[]* argument.
- For the purposes of the exam assume that the *main( )* method must be *public*.
- Improper *main( )* method declaration (or the lack of a *main( )* method) cause a runtime error not a compiler error.
- In the declaration of *main( )*, the order of the keywords *public* and *static* can be switched.

java.lang.Runnable

- The *java.lang.Runnable* interface has a single method
  public void run( );

Interface Implementation

- Interfaces are contracts for what a *class* can do, but they say nothing about the way in which the *class* must do it.
- Interfaces can be implemented by any *class*, from an inheritance tree.
- An interface is like a 100-percent *abstract class*, and is implicitly *abstract* whether you type the *abstract* modifier in the declaration or not, although an *interface* doesn't have a constructor (or get constructed) and an *abstract class* does.
- An *interface* can only have *abstract* methods, no concrete methods are allowed.
- Interfaces are by default *public* and *abstract* – explicit declaration of these modifiers is optional.
- Interfaces can also be declared *strictfp.*
- Interfaces **can have** constants, which are always implicitly *public*, *static* and *final*.
- *interface* constants **cannot be** *transient*.
- *interface* methods cannot be *native*, *strictfp* or *synchronized*.

- *interface* constant declarations of *public*, *static* and *final* are optional in any combination.
- A legal nonabstract implementing *class* has the following properties:
    - It provides concrete implementations for all methods from the *interface*.
    - It must follow all legal override rules for the methods it implements.
    - It must not declare any new checked exceptions for an implementation method.
    - It must not declare any checked exceptions that are broader than the exceptions declared in the *interface* method, but does not have to declare the exceptions of the *interface*.
    - It may declare runtime exceptions on any *interface* method implementation regardless of the *interface* declaration.
    - It must maintain the exact signature and return type of the methods it implements (but does not have to declare the exceptions of the *interface*).
- A *class* implementing an *interface* can itself be *abstract*.
- An *abstract* implementing *class* does not have to *implement* the *interface* methods (but the first concrete subclass must).
- A *class* can *extend* only one *class* (no multiple inheritance), but it can *implement* many interfaces.
- Interfaces can *extend* one or more other interfaces.
- Interfaces cannot *extend* a *class*, or *implement* a *class* or *interface*.
- When taking the exam, verify that *interface* and *class* declarations are legal before verifying other code logic.

Java Operators

- An unassigned reference variable's bits represent *null*.
- There are 12 assignment operators: =, *=, /=, %=, +=, -=, <<=, >>=, >>>=, &=, ^=, |=
- Numeric expressions always result in at least an *int*-sized result never smaller.
- Floating-point numbers are implicitly doubles (64-bits)
- Narrowing a primitive truncates the high-order bits.
- To convert to or from Two's Complement: flip all the bits, then add 1
- Compound assignments (e.g. +=) perform an automatic cast.

String Objects and References

- *String* objects are immutable, and cannot be changed

- When you use a *String* reference variable to modify a *String*:
    - A new S*tring* is created (the old *String* is immutable)
    - The reference variable is set to the new *String*.

Comparison Operators

- Comparison operators always result in a *boolean* value (*true* or *false*)
- There are four comparison operators: >, >=, <, <=
- When comparing characters, Java uses the ASCII or Unicode value of the number as the numerical value.

Equality Operator

- Four types of things can be tested: numbers, characters, *booleans*, reference variables.
- There are two equality operators: == and !=

Arithmetic Operators

- There are four primary operators: add, subtract, multiply, and divide
- The remainder operator returns the remainder of a division.
- When floating-point numbers are divided by zero, they return positive or negative infinity, except when the dividend is also zero, in which case you get NaN.
- When the remainder operator performs a floating-point divide by zero, it will not cause a runtime exception.
- When integers are divided by zero, a runtime *ArithmeticException* is thrown.
- When the remainder operator performs an integer divide by zero, a runtime *ArithmeticException* is thrown.

String Concatenation Operator

- If either operand is a *String*, the + operator concatenates the operands.
- If both operands are numeric, the + operator adds the operands.

Increment / Decrement Operators

- Prefix operator runs before the value is used in the expression.
- Postfix operator runs after the value is used in the expression.
- In any expression, both operands are fully evaluated before the expression is applied.
- *final* variable cannot be incremented or decremented.

Shift Operators

- There are three shift operators: >>, <<, >>>; the first two are signed the last is unsigned.
- Shift operators can only be used on integer types.
- Shift operators can work on all bases of integers (octal, decimal or hexadecimal)
- Except for the unusual cases of shifting an *int* by a multiple of 32 or a *long* by a multiple of 64 (these shifts result in no change to the original values), bits are filled as follows:
    - << fills the right bits with zeros
    - >> fills the left bits with whatever value the original sign bits (leftmost bit) held.
    - >>> fills the left bits with zeros (negative numbers will become positive)
- All bit shift operands are promoted to at least an *int*.
- For *int* shifts > 32 or *long* shifts > 64, the actual shift value is the remainder of the right operand divided by 32 or 64 respectively.

Bitwise Operators

- There are three bitwise operators &, ^, | and a bitwise complement operator ~.
- The & operator sets a bit to 1 if both operand's bits are set to 1 and is therefore a bitwise AND operator.
- The ^ operator sets a bit to 1 if exactly one operand's bit is set to 1 and therefore is a bitwise XOR operator.
- The | operator sets a bit to 1 if at least one operand's bit is set to 1 and therefore is a bitwise OR operator.
- The ~ operator reverses the value of every bit in the single operand.

Ternary (Conditional Operator)

- *someVariable* **=** **(***boolean expression***) ?** *value to assign if true* **:** *value to assign if false*
- Returns one of two values based on whether a boolean expression is *true* of *false*.
- The value after the ? is the "if true return"
- The value after the : is the "if false return"

Casting

- Implicit casting (you write no code) happens when a widening conversion occurs.
- Explicit casting (you write the cast) is required when a narrowing conversion is desired.

- Casting a floating point to an integer type causes all digits to the right of the decimal point to be lost (truncated).
- Narrowing conversion can cause loss of data – the most significant bits (leftmost) can be lost.

Logical Operators

- There are five logical operators: ^, &, |, &&, ||
- Logical operators work with two expressions that must resolve to *boolean* values.
- The && and & (AND) operators return *true* only if both operands are *true*.
- The | and || (OR) operators return *true* if either (or both) operands are *true*.
- The ^ (XOR) operator return *true* if either (but not both) operands are *true*.
- The && and || operators are known as short-circuit operators.
- The && operator does not evaluate the right operand if the left operand is *false*.
- The || operator does not evaluate the right operand if the left operand is *true*.
- The &, | and ^ operators always evaluate both operands.

Passing Variables into Methods

- Methods can take primitive and / or object references as arguments.
- Method arguments are always copies of either primitive variables or reference variables.
- Method arguments are never actual objects (they can be references to objects).
- In practice, a primitive argument is a completely detached copy of the original primitive.
- In practice, a reference argument is another copy of a reference to the original object.

Writing Code using if and switch Statements

- The only legal argument to an *if* statement is a *boolean*, so the *if* test can be only on an expression that resolves to a *boolean* or a *boolean* variable.
- Watch out for *boolean* assignments (=) that can be mistaken for *boolean* equality (==) tests.
- Curly braces are optional for *if* statement that have only one statement, but watch out for misleading indentation.
- *switch* statements can be evaluate only the *byte*, *short*, *int* and *char* data types, i.e. only integers 32-bits or less.

- When switching on a variable smaller than an *int* all the case arguments must be able to be cast to the smaller variable type without affecting their value.

```
byte g = 2; // max byte value is 127
switch (g) {
        case 23:
                …
                break;
        case 128:
                …
                break;  // compiler error:
                        // possible loss of precision
                        // found: int
                        // required: byte
}
```

- The *while* condition of a *do-while* requires a semicolon at the end.

```
do {
        …
} while ( … ) ;
```

- The *case* argument must be a literal or *final* variable. You cannot have a *case* that includes a non-final variable, or a range of values.
- If the condition in a *switch* statement matches a *case* value, execution will run through all code in the *switch* following the matching *case* statement until a *break* or the end of the *switch* statement is encountered. In other words, the matching *case* is just the entry point into the *case* block, but unless there's a *break* statement, the matching *case* is not the only *case* code that runs.
- The *default* keyword should be used in a *switch* statement if you want to execute some code when none of the *case* values match the conditional value.
- The *default* block can be located anywhere in the *switch* block so if no *case* matches, the *default* block will be entered, and if the *default* does not contain a *break*, then code will continue to execute (fall-through) to the end of the *switch* or until the *break* statement is encountered.

Writing Code Using Loops

- A *for* statement does not require any arguments in the declarations, but has three parts: declaration and / or initialization, *boolean* evaluation, and the iteration expression.
- If a variable is incremented or evaluated within a *for* loop, it must be declared before the loop, or within the *for* loop declaration.
- A variable declared (not just initialized) within the *for* loop declaration cannot be accessed outside the *for* loop.
- Both the initialize and iteration expressions can have more than one statement each separated by a comma, all variables initialized this way must be the same type.
- The conditional expression can not have more than one statement.
- The *do-while* loop will enter the body of the loop at least once, even if the test condition is not met.

Using break and continue

- An unlabeled *break* statement will cause the current iteration of the innermost looping construct to stop and the next line of code following the loop to be continued.
- An unlabeled *continue* statement will cause the current iteration of the innermost loop to stop, and the condition of that loop to be checked, and if the condition is met, perform the loop again.
- If the *break* statement or the *continue* statement is labelled, is will cause similar action to occur on the labelled loop, not the innermost loop, i.e.

```
outer:
    while(true) {
        do {
            System.out.println("Hello");
            break outer;
        } while (false) ;
    }
```

- If a *continue* statement is used in a *for* loop, the iteration statement is executed, and the condition is checked again.

Catching an Exception Using *try* and *catch*

- Exceptions come in two flavours: checked and unchecked

- Checked exceptions include all subtypes of *Exception*, excluding classes that extend *RuntimeException*.
- Checked exceptions are subject to the handle or declare rule; any method that might throw a checked exception (including methods that invoke methods that can throw a checked exception) must either declare the exception using the *throws* keyword, or handle the exception with an appropriate *try - catch*.
- Subtypes of *Error* or *RuntimeException* are unchecked, so the compiler doesn't enforce the handle or declare rule. You're free to handle them, and your free to declare them, but the compiler doesn't care one way or the other.
- If you use an optional *finally* block, it will always be invoked, regardless of whether an exception in the corresponding *try* is thrown or not, and regardless of whether a thrown exception is caught or not.
- The only exception to the finally-will-always-be-called rule is that a *finally* will not be invoked if the JVM shuts down. That could happen if code from the *try* or *catch* blocks calls *System.exit(int)*, in which case the JVM will not start your *finally* block.
- Just because *finally* is invoked does not mean it will complete. Code in the *finally* block could itself raise an exception or issue a *System.exit(int)*.
- Uncaught exceptions propagate back through the call stack, starting from the method where the exception is thrown and ending with either the first method that has a corresponding *catch* for that exception type or a JVM shutdown (which happens if the exception gets to *main( )*, and *main( )* is "ducking" the exception by declaring it).
- You can create your own exceptions, normally by extending *Exception* or one of its subtypes. Your exception will then be considered a checked exception, and the compiler will enforce the handle-or-declare rule for that exception.
- All *catch* blocks must be ordered from most specific to most general. For example, if you have a *catch* clause for *IOException* and *Exception*, you must put the catch for *IOException* first. Otherwise, the *IOException* would be caught by *catch(Exception e)*, because a *catch* argument can catch the specified exception or any of its subtypes. The compiler will stop you from defining *catch* clauses that can never be reached (because it sees that the more specific exception will be caught first by the more general *catch*).

Working with the Assertion Mechanism

- Assertions give you a way to test your assumptions during development and debugging.

- Assertions are typically enabled during testing but disabled during deployment.
- *assert* statements have the following syntax:
  assert (<boolean expression>);
  or
  assert (<boolean expression>) : "<string message>" ;
- You can use *assert* as a keyword or an identifier, but not both together. To compile code that uses *assert* as an identifier (for example, a method name), use the *–source 1.3* command-line flag to *javac*. To compile code that uses *assert* as a keyword use the *–source 1.4* flag to *javac*.
- Assertions are disabled at runtime (and compile-time) by default. To enable them, use a command-line flag *–ea* or *–enableassertions*.
- You can selectively disable assertions using the *–da* or *–disableassertions* flag.
- If you enable or disable assertions using the flag without any arguments, you're enabling or disabling assertions in general. You can combine enabling and disabling switches to have assertions enabled for some classes and or packages, but not others.
- You can enable or disable assertions in the system classes with the *–esa* or *–dsa* flags.
- You can enable and disable assertions on a class-by-class basis, using the following syntax:
  java -ea -da:MyClass TestClass
- You can enable and disable assertions on a package basis, and any package you specify also includes any subpackages (packages further down the directory hierarchy)
  java -ea -da:mypackage.subpackage TestClass
- **Do not use** assert expressions to validate arguments to *public* methods.
- **Do not use** assert expressions that cause side effects. Assertions aren't guaranteed to always run, so you don't want behaviour that changes depending on whether assertions are enabled.
- **Do use** assertions to validate that a particular code block will never be reached, by using: *assert false;* this will throw an assertion error immediately if the assert statement is executed.

Encapsulation

- The goal of encapsulation is to hide an implementation behind an interface (or API).
- Encapsulated code has two features:

    - Instance variables are kept protected (usually with the *private* modifier)
    - Getter and setter methods provide access to instance variables
- IS-A refers to inheritance
- IS-A is expressed with the keyword *extends*
- HAS-A means an instance of one *class* "has a" reference to an instance of another *class*.

Overriding and Overloading

- Instance methods can be overridden or overloaded; constructors and *static* methods can be overloaded but not overridden.
- *abstract* methods must be overridden by the first concrete (nonabstract) subclass.
- With respect to the method it overrides, the overriding method
    - Must have the same argument list
    - Must have the same *return* type
    - Must not have a more restrictive access modifier
    - May have a less restrictive access modifier
    - Must not throw new or broader checked exceptions
    - May throw fewer or narrower checked exceptions, or any unchecked exception
- *final* methods cannot be overridden, and therefore cannot be *abstract*.
- Only inherited methods may be overridden.
- A subclass uses *super.overriddenMethodName( )* to call the superclass version of an overridden method.
- Overloading means using the same method name, but with different arguments.
- Overloaded methods
    - Must have different argument lists
    - May have different return types, as long as the argument lists are also different
    - May have different access modifiers
    - May throw different exceptions
- Methods from a superclass can be overloaded in a subclass.
- Polymorphism applies to overriding, not to overloading.
- Object type (of the calling instance) determines which overridden method is used at runtime.
- Reference type (of the method parameters) determines which overloaded method will be used at compile time.

Instantiation and Constructors

- Objects are constructed:
  - You cannot create a new object without invoking a constructor
  - Each superclass in an object's inheritance tree will have one of its constructors called
- Every *class*, even *abstract* classes, has at least one constructor.
- Constructors must have the same name as the *class*.
- Constructors do not have a *return* type. If there is a *return*, then it is simply a method with the same name as the *class*, and not a constructor.
- Constructors execution occurs as follows:
  - The constructor calls its superclass constructor, which call its superclass constructor, and so on all the way up to the *Object* constructor.
  - The *Object* constructor executes and then returns to the calling constructor, which runs to completion and then returns to its calling constructor, and so on back down to the constructor of the actual instance being created.
- Constructors can use any access modifiers (even *private*).
- The compiler will create a default constructor if you don't create any constructors in your class.
- The default constructor is a *public* no-argument constructor with a no-argument call to *super( )*.
- The first statement of every constructor must be a call to either *this( )* (an overloaded constructor) or *super( )*.
- The compiler will add a call to *super( )* if you do not, unless you have already put in a call to *this( )*.
- Instance methods and variables are only accessible after the *super( )* constructor runs.
- *abstract* classes have constructors that are called when a concrete subclass is instantiated.
- Interfaces do not have constructors.
- If your superclass does not have a no-argument constructor, you must create a constructor and insert a call to *super( )* with arguments matching those of the superclass constructor.
- Constructors are never inherited, thus they cannot be overridden.
- A constructor can be directly invoked only by another constructor, using a call to *super( )* or *this( )*.
- Issues with calls to *this( )*:

- May appear only as first statement in a constructor
- The argument list determines which overloaded constructor is called.
- Constructors can call constructors can call constructors, and so on, but sooner or later one of them better call *super( )* or the stack will explode.
- *this( )* and *super( )* cannot be in the same constructor. You can have one or the other but never both.

Return Types

- Overloaded methods can change *return* types; overridden methods cannot.
- Object reference *return* types can accept *null* as a *return* value.
- An array is a legal *return* type, both to declare and *return* as a value.
- For methods with primitive *return* type, any value that can be implicitly converted to the *return* type can be returned.
- Nothing can be returned from a *void* method, but you can *return* nothing, by using the *return* keyword on its own.
- For methods with an object reference *return* type, a subclass of that type can be returned.
- For methods with an interface *return* type, any implementer of that interface (or a subinterface) can be returned.

Inner Classes

- A "regular" inner *class* is declared inside the curly braces of another *class*, but outside any method or other code block.
- An inner *class* is a full-fledged member of the enclosing (outer) *class*, so it can be marked with an access modifier as well as the *strictfp, abstract* or *final* modifiers, but of course never *abstract* and *final* together.
- An inner *class* instance shares a special relationship with an instance of the enclosing *class*. This relationship gives the inner *class* access to all of the outer *class*' members, including those marked *private.*
- To instantiate an inner *class*, you must have a reference to an instance of the outer *class*.
- From code within the enclosing *class*, you can instantiate the inner *class* using only the name of the inner *class*, as follows:
  MyInner mi = new MyInner( );

- From code outside the enclosing *class*' instance methods, you can instantiate the inner *class* only by using both the inner and outer *class* names, and a reference to the outer *class* as follows:
  MyOuter mo = new MyOuter( );
  MyOuter.MyInner inner = mo.new MyInner( );
- From code within the inner *class*, the keyword *this* holds a reference to the inner *class* instance. To reference the *outer-this* (in other words, the instance of the outer *class* that this inner instance is tied to) precede the keyword *this* with the outer *class* name as follows:
  MyOuter.this;

Method-Local Inner Classes

- A method-local inner *class* is defined within a method of the enclosing *class*.
- For the inner *class* to be used, you must instantiate it, and that instantiation must happen within the same method, but after the *class* definition code.
- A method-local inner *class* cannot use variables declared within the method (including parameters) unless those variable are marked *final.*
- The only modifiers you can apply to a method-local inner *class* are *strictfp, abstract* and *final* (Never both *final* and *abstract* at the same time).

Anonymous Inner Classes

- Anonymous inner classes have no name, and their type must be either a subclass of the named type or an implementer of the named *interface*.
- An anonymous inner *class* can be recognized because it has an pair of braces containing the *class* definition before the semicolon. An anonymous inner *class* is always created as part of a statement, so don't forget to close the statement after the *class* definition, with a curly brace and semicolon.

```
Runnable r = new Runnable( ) {
        public run( ) { ... }
};
```

- Because of polymorphism, the only method you can call on an anonymous inner *class* reference are those defined in the reference variable *class* (or *interface*).
- An anonymous inner *class* can *extend* one *class*, *or implement* one *interface*. Unlike non-anonymous classes (inner or otherwise), an anonymous inner *class* cannot do both. In

other words, it cannot both *extend* a *class* and *implement* an *interface*, nor can it *implement* more than one *interface*.

- An argument-local inner *class* is declared, defined, and automatically instantiated as part of a method invocation. The key to remember is that the *class* is being defined within a method argument, so the syntax will end the *class* definition with a curly brace, followed by a closing parenthesis to end the method call, followed by a semicolon to end the statement: *});*

Static Nested Classes

- Static nested classes are inner classes marked with the *static* modifier.
- Technically, a static nested class is *not* an inner class, but instead considered a top-level nested *class*.
- Because the nested *class* is *static*, it does not share any special relationship with an instance of the outer *class*. In fact, you don't need an instance of the outer *class* to instantiate a *static* nested *class*.
- Instantiating a *static* nested *class* requires using both the outer and nested *class* names as follows:
  MyOuter.StaticInner sn = new MyOuter.StaticInner( );
- A *static* inner *class* cannot access nonstatic members of the outer *class*, since it does not have an implicit reference to any outer instance (in other words, the nested *class* instance does not get an *outer-this* reference.

Overriding *hashCode( )* and *equals( )*

- The critical methods in class *Object* are *equals( ), finalize( ), hashCode( )*, and *toString( ).*
- *equals( ), hashCode( )*, and *toString( )* are *public* and *finalize( )* is *protected*.
- Fun facts about *toString( )*
  - Override *toString( )* so that *System.out.println( )* or other methods can see something useful.
  - Override *toString( )* to return the essence of your object's state.
- Use == to determine if two reference variables refer to the same object.
- Use *equals( )* to determine if two objects are meaningfully equivalent.
- If you don't override *equals( )*, your objects won't be useful hashtable / hashmap keys.
- If you don't override *equals( )*, two different objects can't be considered the same.

- Strings and wrappers override *equals( )* and make good hashtable / hashmap keys.
- When overriding *equals( )*, use the *instanceof* operator to be sure you're evaluating an appropriate *class*.
- When overriding *equals( )*, compare the object's significant attributes.
- Highlights of the *equals( )* contract:
    - Reflexive: *x.equals(x)* is *true*
    - Symmetric: If *x.equals(y)* is *true*, then *y.equals(x)* must be *true*
    - Transitive: If *x.equals(y)* is *true*, and *y.equals(z)* is *true*, then *x.equals(z)* is *true*
    - Consistent: Multiple calls to *x.equals(y)* will return the same result
    - Null: If *x* is not *null* then *x.equals(null)* is *false*
- If *x.equals(y)* is *true*, then *x.hashCode( ) == y.hashCode( )* must be *true*.
- If you override *equals( )*, override *hashCode( )*; it is also good to *implement* the *serializable interface* as *HashMap* and *Hashtable* are both themselves *serializable*.
- Classes *HashMap*, *Hashtable*, *LinkedHashMap*, and *LinkedHashSet* use hashing.
- A **legal** *hashCode( )* override compiles and runs.
- An **appropriate** *hashCode( )* override sticks to the contract.
- An **efficient** *hashCode( )* override distributes keys randomly across a wide range of buckets.
- To reiterate: if two objects are equal, their hashcodes must be equal.
- It's legal for a *hashCode( )* method to return the same value for all instances, although in practice it's very inefficient.
- Highlights of the *hashCode( )* contract:
    - Consistent: Multiple calls to *x.hashCode( )* return the same integer.
    - If *x.equals(y)* is *true*, then *x.hashCode( ) == y.hashCode( )* must be *true*.
    - If *x.equals(y)* is *false*, then *x.hashCode( ) == y.hashCode( )* can be either *true* or *false*, but *false* will tend to create better efficiency.
- *transient* variables aren't appropriate for *equals( )* and *hashCode( )*.

Collections

- Common collection activities include adding objects, removing objects, verifying object inclusion, retrieving objects, and iterating.
- Three meanings for "collection":
    - collection – Respresents the data structure in which objects are stored.

  - *Collection – java.util.Collection* is the interface from which *Set* and *List* extend.
  - *Collections* – A class that holds *static* collection utility methods
- Three basic flavours of collection include *List*, *Set* and *Map*:
  - *List* – Ordered, duplicates allowed, with an index
  - *Set* – May or may not be ordered and / or sorted, duplicates not allowed
  - *Map* – May or may not be ordered and / or sorted, uses a key, duplicate keys are not allowed
- Four basic subflavours of collections included Sorted, Unsorted, Ordered, and Unordered.
- Ordered means iterating through a collection in a specific, non-random order.
- Sorted means iterating through a collection in a natural sorted order.
- Natural means alphabetic, numeric, or programmer-defined, which ever applies.
- Key attributes of common collection classes:
  - *ArrayList* – Fast iteration and fast random access
  - Vector – Like a somewhat slower *ArrayList*, mainly due to its *synchronized* methods
  - *LinkedList* – Good for adding and removing elements from the ends (i.e. stacks and queues) and the middle
  - *HashSet* – Assures no duplicates, provides no ordering
  - *LinkedHashSet* – No duplicates, iterates by insertion order or last accessed
  - *TreeSet* – No duplicates, iterates in natural sorted order
  - *HashMap* – Fastest updates (key/value pairs), allows one *null* key , many *null* values
  - *Hashtable* – Like a slower *HashMap* (as with *Vector*, due to its *synchronized* methods). No *null* values or *null* key allowed.
  - *LinkedHashMap* – Faster iterations, iterates by insertion order or last accessed, allows one *null* key, many *null* values.
  - *TreeMap* – A sorted map, in natural order

Garbage Collection

- In Java, garbage collection provides some automated memory management.
- All objects in Java live on the heap.
- The heap is also known as the garbage collectable heap.

- The purpose of garbage collecting is to find and delete the objects that can't be reached
- Only the JVM decides exactly when to run the garbage collector.
- You (the programmer) can only recommend when to run the garbage collector.
- You can't know the G.C. algorithm; maybe it uses mark and sweep, maybe it's generational and / or iterative.
- Objects must be considered eligible before they can be garbage collected.
- An object is eligible when no live thread can reach it.
- To reach an object, a live thread must have a live reachable reference variable to that object.
- Java applications can run out of memory.
- Islands of objects can be garbage collected, even through they have references,
- To reiterate: garbage collection can't be forced.
- Request garbage collection with *System.gc( );*
- *class Object* has a *finalize( )* method.
- The *finalize( )* method is guaranteed to run once and only once before the garbage collector deletes an object.
- Since the garbage collector makes no guarantees, *finalize( )* may never run.
- You can uneligibilize an object from within *finalize( )*.
- It is not normally considered a good ides to override the *finalize( )* method as it may never run.

Creating, Instantiating, and Starting New Threads

- Threads can be created by extending *Thread* and overriding the *public void run ( )* method.
- Thread objects can also be created by calling the *Thread* constructor that takes a *Runnable* argument. The *Runnable* object is said to be the target of the thread.
- You can call *start( )* on a *Thread* object only once. If *start( )* is called more than once on a *Thread* object, it will throw a *RuntimeExpection*.
- The *isAlive( )* method *return true* if *start( )* has been called and the *Thread* has not yet died.
- It is legal to create many *Thread* objects using the same *Runnable* object as the target.

- When a *Thread* object is created, it does not become a thread of execution until its *start( )* method is invoked. When a *Thread* object exists but hasn't been started, it is in the **new** state and is not considered **alive**.

Transition Between Thread States

- Once a new thread is started, it will always enter the **runnable** state.
- The thread scheduler can move a thread back and forth between the **runnable** state and the **running** state.
- Only one thread can be **running** at a time, although many threads may be in **runnable** state.
- There is no guarantee that the order in which threads were started determines the order in which they'll run.
- There' no guarantee that threads take turns in any fair way. It's up to the thread scheduler, as determined by the particular virtual machine implementation. If you want a guarantee that your threads will take turns regardless of the underlying JVM, you should use the *sleep( )* method. This prevents one thread from hogging the running process while another thread starves.
- A running *Thread* may enter a **blocked / waiting** state by a *wait( ), sleep( ),* or *join( )* call.
- A running *Thread* may enter a **blocked / waiting** state because it can't acquire the lock for a *synchronized* block of code.
- When the **sleep** or **wait** is over, or an object's lock becomes available, the *Thread* can only reenter the **runnable** state.
- A **dead** *Thread* cannot be started again.

Sleep, Yield and Join

- **Sleeping** is used to delay execution for a period of time, and no locks are released when a *Thread* goes to sleep.
- A **sleeping** thread is guaranteed to sleep for at least the time specified in the argument to the sleep method (unless it's interrupted), but there is no guarantee as to when the newly awakened *Thread* will actually return to running.
- The *sleep( )* method is a *static* method that sleeps the currently executing thread. One *Thread* cannot tell another *Thread* to sleep.

- The *setPriority( )* method is used on *Thread* objects to give threads a priority of between 1 (low) and 10 (high), although priorities are not guaranteed, and not all JVMs use a priority range of 1-10.
- If not explicitly set, a thread's priority will be the same priority as the thread that created it (in other words, the *Thread* executing the code that creates the new *Thread*)
- The *yield( )* method may cause a running thread to back out if there are runnable threads of the same priority. There is no guarantee that this will happen, and there is no guarantee that when the *Thread* backs out there will be a different *Thread* selected to run. A *Thread* might *yield( )* and then immediately reenter the running state.
- The closest thing to a guarantee is that at any given time, when a *Thread* is running it will usually not have a lower priority than any *Thread* in the runnable state. If a low-priority *Thread* is running when a high-priority *Thread* enters runnable, the JVM will usually pre-empt the running low-priority *Thread* and put the high-priority *Thread* in.
- When one *Thread* calls the *join( )* method of another *Thread*, the currently running *Thread* will wait until the *Thread* it joins with has completed. Think of the *join( )* method as saying "Hey thread, I want to join on to the end of you. Let me know when you're done, so I can enter the **runnable** state."

Concurrent Access Problems and *synchronized* Threads

- *synchronized* methods prevent more than one thread from accessing an object's critical method code.
- You can use the *synchronized* keyword as a method modifier, or to start a *synchronized* block of code.

  synchronized (obj) {

  …

  }
- To synchronize a block of code (in other words, a scope smaller than the whole method), you must specify an argument that is the object whose lock you want to synchronize on.
- While only one thread can be accessing *synchronized* code of a particular instance, multiple threads can still access the same object's unsynchronized code.
- When an object goes to **sleep**, it takes its locks with it.
- *static* methods can be *synchronized*, using the lock from the *java.lang.Class* instance representing that *class*.

Communicating with Objects by Waiting and Notifying

- The *wait( )* method lets a *Thread* say, "there's nothing for me to do here, so put me in your waiting pool and notify me when something happens that I care about." Basically a *wait( )* call means "wait me in your pool", or "add me to your waiting list"
- The *notify( )* method is used to send a signal to one and only one of the threads that are waiting in the same object's waiting pool.
- The method *notifyAll( )* works in the same way as *notify( )*, only it sends the signal to all of the threads waiting on the object.
- All three methods: *wait( ), notify( ),* and *notifyAll( )* must be called from within a *synchronized* context. A *Thread* invokes *wait( ), notify( )* and *notifyAll( )* on a particular object, and the thread must currently hold the lock on that object.

```
synchronize void doSomething( ) {
        while (spareResources <= 0) {
                wait( );
        }
        spareResources--;
        // use resource
        spareResources++;
        notifyAll( );
}
```

Deadlocked Threads

- Deadlocking is when thread executing grinds to a halt because the code is waiting for locks to be removed from objects.
- Deadlocking can occur when a locked object attempts to access another locked object that is trying to access the first locked object. In other words, both threads are waiting for each other's locks to be released; therefore, the locks will never be released.
- Deadlocking is bad, never do it.

APPENDIX A: KEYWORDS

Access Modifiers

- ***private*** – accessible only from within its own class
- ***protected*** – accessible only to classes in the same package or subclasses of the class
- ***public*** – accessible from any other class

Class, Method, and Variable Modifiers

- ***abstract*** – declare a class that cannot be instantiated, or a method that must be implemented by a nonabstract subclass
- ***class*** – specify class
- ***extends*** – indicate the superclass a subclass is extending
- ***final*** – impossible to extend a class, override a method, or reinitialize a variable
- ***implements*** – indicates the interfaces that a class will implement
- ***interface*** – specify interface
- ***native*** – indicates a method is written in a platform-dependent language
- ***new*** – instantiate an object by invoking constructor
- ***static*** – method or variable belongs to class (not instant)
- ***strictfp*** – used in front of a method or class to indicate that floating-point numbers will strictly follow the Java Specification for floating point calculation (i.e. only 64-bit) and will no more accurate. *strictfp* therefore provides repeatability for different platforms.
- ***synchronised*** – indicates a method can be accessed by only one thread a time
- ***transient*** – prevents fields from being serialized, as transient fields are always skipped when objects are serialized
- ***volatile*** – used on variables that may be modified simultaneously by other threads. This warns the compiler to fetch them fresh each time, rather than caching them in registers. This also inhibits certain optimisations that assume no other thread will change the values unexpectedly. Since other threads cannot see local variables, there is never any need to mark local variables volatile.

Flow Control

- ***break*** – exits from the block of code in which it resides
- ***case*** – executes a block of code, dependent on what the switch tests for

- ***continue*** – stops the rest of the code following this statement from executing in a loop and then begins the next iteration of the loop, i.e. The break statement is used to exit from a loop or switch statement, while the continue statement is used to skip just the current iteration and continue with the next
- ***default*** – executes this block of code if none of the switch-case statements match
- ***do*** – executes a block of code one time, then, in conjunction with the while statement, it performs a test
- ***else*** – executes an alternative block of code if an if test is false
- ***for*** – used to perform a conditional loop for a block of code
- ***if*** – used to perform a logical test for true or false
- ***instanceof*** – determines whether an object is an instance of a class superclass, or interface
- ***return*** – returns from a method without executing any code that follows the statement and can optionally return a variable
- ***switch*** – indicates the variable to be compared with the different case statements
- ***while*** – executes a block of code repeatedly while a certain condition is true

Error Handling

- ***catch*** – declares the block of code used to handle an exception
- ***finally*** – block of code, usually following a try-catch statement, which is executed no matter what program flow occurs when dealing with an exception
- ***throw*** – used to pass an exception up to the method that called this method
- ***throws*** – indicates the method will pass an exception to the method that called it
- ***try*** – block of code that will be tried, but may cause an exception
- ***assert*** – evaluates a conditional expression to verify the programmer's assumption, throws *AssertionError* if condition "assumption" not true.

Package Control

- ***import*** – statement to import packages or classes into code
- ***package*** – specifies to which package all classes in a source file belong

Primitives

- 64-bit

    - ***double*** – a 64-bit floating-point number (signed)
    - ***long*** – a 64-bit integer (signed)
- 32-bit
    - ***float*** – a 32-bit floating-point number (signed)
    - ***int*** – a 32-bit integer (signed)
- 16-bit
    - ***char*** – a 16-bit single Unicode character (unsigned)
    - ***short*** – a 16-bit integer (signed)
- 8-bit
    - ***byte*** – an 8-bit integer (signed)
- < 8-bit
    - **boolean** – a value indicating true or false

Variable Keywords

- ***super*** – reference variable referring to the immediate superclass
- ***this*** – reference variable referring to the current instance of an object

Void Return Type Keyword

- ***void*** – indicates no return type for a method

Unused Reserved Words

- ***const*** – do not used this the correct keyword in Java is static
- ***goto*** – considered bad programming so not implemented in Java

APPENDIX B: QUICK SUMMARY

Primitive Variables' Size

64-bit:
- double
- long

32-bit:
- float
- int

16-bit:
- char
- short

8-bit:
- byte

< 8-bit:
- boolean

Primitive Integer Variables' Range

$-2^{(bits-1)}$ to $2^{(bits-1)}-1$

Member Visibility

private = class
default = package
protected = package + subclasses
public = all

Class Visibility

default = package
public = all

3 Types of Array Declaration

```
String[] days = {"su", "mo"};
String []days = new String [] {"su","mo"};
String days[] = new String[2];
```

2 Main Array Exceptions

NullPointerException
ArrayIndexOutOfBoundsException

Class Definitions

Local / method / automatic / stack variables don't get default values so must be initialized before use.

final reference variables must be initialized before the constructor completes.

Polymorphism does not apply to overridden *static* methods. An instance reference variable used to call a *static* method is converted into a *class* reference by the compiler and the instance variable is not used.

abstract methods must be implemented by a subclass, so they must be inheritable. For that reason *abstract* **methods cannot be**:

private.
final.
synchronized.
native.
strictfp

but *abstract* **classes can be** *strictfp.*

Interfaces can have constants, which are always implicitly *public*, *static* and *final*; explicit declaration of these modifiers is optional.

Interfaces can extend one or more interfaces.

Character Formats

To specify a character using its Unicode value use a backslash followed by an 8-bit octal value *'\000'* through *'\377'* or by *u* or *U* followed by a four-digit hexadecimal value *'\u0000'* (or *'\U0000'*) through *'\uffff'* (or *'\UFFFF'*). The four digit hexadecimal value can be used to specify the full range of Unicode characters.

Number Formats

octal	leading zero (0127)
decimal	no leading zeros (1245)
hexadecimal	leading 0X or 0x (0XCAFE or 0xCAFE)

Numeric expressions always result in at least an *int* sized (32-bit) result never smaller.

Narrowing truncates high order bits (left-most bits).

Two's Compliment

Flip the bits and add 1
Highest order bit is sign:

0 = +ve
1 = -ve

Shifts

Shifts can only be used on integers

>>,	right shift
<<,	left shift
>>>	unsigned right shift (zero-filled right shift)

Division by Zero

1.2f/0 = Float.POSITIVE_INFINITY
-2.2d/0 = Double.NEGATIVE_INFINITY
0d/0 = Double.NaN
1l/0 = ArithmeticException

Bitwise Non-Lazy Boolean Operators

& AND
| OR
^ XOR
~ Compliment

Loops and *if* statements

(boolean expression) ? value if *true* : value if *false*

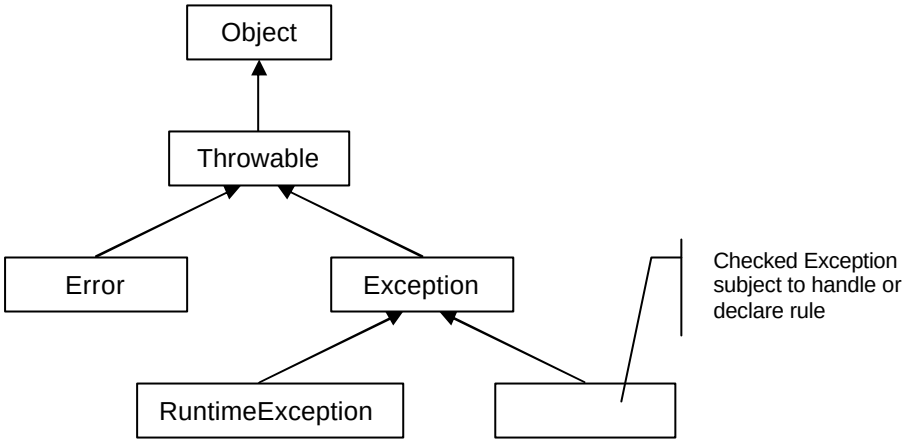
switch statements can only be evaluated by integers 32-bits or less, such as: *bytes*, *short*, *char*, and *int*.

An *else* clause belongs to the innermost *if* statement to which it might possible belong, in other words the closest preceding *if* that doesn't have an *else*.

Labelled *break* / *continue* statements

```
outer:
        do {
                if ( … ) {
                        …
                        continue outer;
                }
        } while ( … ) ;
```

Exceptions



The only exception to the finally-will-always-be-called rule is that a finally will not be invoked if the JVM shuts down. That could happen if code from the try or catch blocks call *System.exit(int);*

printStackTrace( ) is a useful method provided by *Throwable* that prints the stack trace from where the exception occurred.

Asserts

assert (<boolean expression>) ;

assert (<boolean expression>) : " <message> " ;

Classes

A class literal is formed by appending *.class* to the name of a primitive or reference type. It evaluates to the class descriptor of the reference type or class descriptor of the primitive type's wrapper class.

A subclass uses *super.overiddenMethodName( )* to call an overridden superclass method whereas an inner-class uses *OuterClassType.this.overiddenMethodName( )* to call an overridden outer-class method.

An *abstract class* has a constructor (called during object creation) but an *interface* does not.

Overloaded methods can change return types; overridden methods cannot.

From code outside an enclosing class' instance methods, you can instantiate an inner class only by using both the inner and outer class names, and a reference to the outer class.

MyOuter mo = new MyOuter( );
MyOuter.MyInner mi = mo.new MyInner( );

A method-local inner *class* cannot use variables declared within the method (including parameters) unless these variable are marked *final*.

Technically, a *static* nested *class* is not an inner *class*, but instead considered a top-level nested *class*.

A *static* inner *class* cannot access nonstatic members of the outer *class*, since it does not have an implicit reference to any outer instance.

static nested classes are declared as follows:

MyOuter.MyStaticInner msi = new MyOuter.MyStaticInner( ) ;

Class Modifiers

normal class:
- public
- abstract
- final
- strictfp

inner class:
- public
- protected
- private
- static
- abstract
- final
- strictfp

method-local inner class:
- abstract
- final
- strictfp

Anonymous inner classes

```
Runnable r = new Runnable( ) {
    public void run( ) { … }
} ;
```

java.lang.Object

If you don't override *equals( )* than an object won't make a good hastable / hashmap key.

If *x.equals(y)* is *true*, then *x.hashCode( ) == y.hashCode( )* is *true*.

If *x.equals(y)* is *false*, then *x.hashCode( ) == y.hashCode( )* can be *true* or *false*, but *false* will tend to create better efficiency.

If *x.hashCode( ) != y.hashCode( )* is *true*, then *x.equlas(y)* is *false*.

transient variables aren't appropriate for *equals( )* and *hashCode( )*.

Collections

java.util.Collection is an interface which *Set* and *List extend*.

java.util.Collections is a *class* that holds *static* collection utility methods.

Vector	synchronized *ArrayList*
Hashtable	synchronized *HashMap* and no null keys or values
ArrayList	fast iteration and fast random access
LinkedList	good for adding and removing elements from ends and middle
HashSet	no duplicates, no ordering
LinkedHashSet	no duplicates, iterates by insertion order or last accessed
TreeSet	no duplicates, iterates by natural sorted order
HashMap	fastest updates, one *null* key many *null* values
LinkedHashMap	faster iterations, iterates by insertion order of last accessed, one *null* key many *null* values
TreeMap	sorted map, in natural order

*Linked**	iterates by insertion order
**Map*	key-value pairs, one *null* key, many *null* values
*Tree**	sorted, iterates in natural order
**Set*	no duplicates
Hash	uses more memory for faster access
**List*	order

Garbage Collection

To request garbage collection use *System.gc( );*

The *finlize( )* method is guaranteed to run once and only once before the garbage collection deletes an object.

Reference Classes

StrongReference	normal reference
SoftReference	Used for memory sensitive caches but may stay for a while after reference dropped
WeakReference	Used for memory sensitive caches but quicker to be removed from memory than *SoftReference*
PhantomReference	A phantom object is one that has been finalized, but whose memory has not yet been made available for another object.

Threads

Threads are created by using

extends Thread

or
implements Runnable
new Thread (Runnable);

Thread constructors:
Thread( )
Thread(Runnable target)
Thread(Runnable target, String name)
Thread(String name)
Thread(ThreadGroup group, Runnable target, String name)
Thread(ThreadGroup group, String name)

Methods from *java.lang.Thread* class

public static void sleep(long millis) throws InterruptedException
public static void yield( )
public final void join( )
public final void setPriority(int newPriority)

Methods from *java.lang.Object* class

public final void wait( )
public final void wait(long timeout)
public final void notify( )
public final void notifyAll( )

You can call *start( )* on a *Thread* object only once. If *start( )* is called more than once on a *Thread* object, it will throw a *RuntimeException*. The *isAlive( )* method can test whether *start( )* has been called and the *Thread* has not yet died. Therefore use *start( )* as follows:

```
Thread t = new Thread( );
if(!t.isAlive( )) {
    t.start( );
}
```

The *sleep( )* is a *static* method that will sleep the currently executing *Thread*. One *Thread* cannot tell another *Thread* to sleep.

The *setPriority( )* method is used on *Thread* objects to give Threads a priority between 1 (low) and 10 (high), although priorities are not guaranteed, and not all JVMs use a priority range of 1-10.

A *Thread*'s implicit priority is the same as its creator.

The *yield( )* method may cause a running *Thread* to back out if there are runnable threads of the same priority. There is no guarantee that this will happen, and there is no guarantee that when the *Thread* backs out there will be a different *Thread* selected to run. A *Thread* might *yield( )* and immediately re-enter the running state. Although, usually, if a low-priority *Thread* is running when a high-priority *Thread* enters runnable, the JVM will pre-empt the running low-priority *Thread* and put the high-priority *Thread* in the running state.

When one *Thread* calls the *join( )* method of another *Thread* the currently running *Thread* will wait until the *Thread* if joins with has completed.

To synchronize code either use the *synchronized* keyword with a method or use the *synchronized(obj) {… }* block.

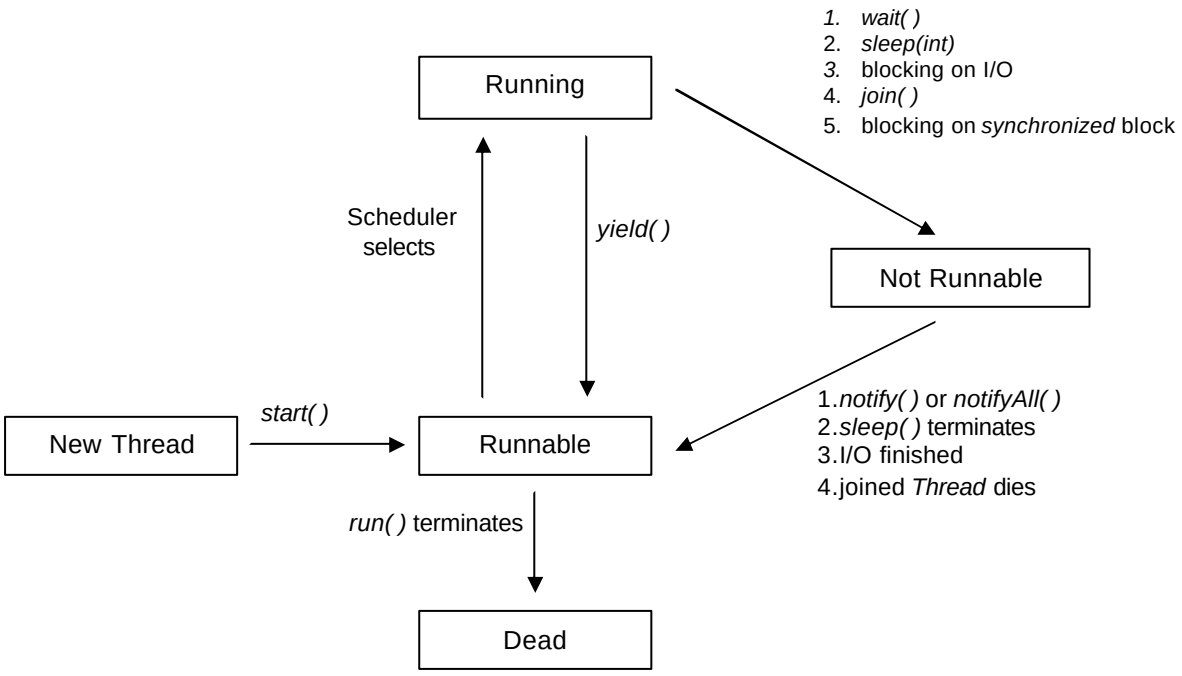
wait( )	put me in wait pool until *notify( )* / *notifyAll( )*
wait(long timeout)	put me in wait pool until *notify( )* / *notifyAll( )* or timeout up
notify( )	wake up one *Thread* from the waiting pool
notifyAll( )	wake up all Threads from the waiting pool

All three methods: *wait( ), notify( ),* and *notifyAll( )* must be called from within a *synchronized* context.

```
synchronized void doSomething( ) {
        while(spareResource <=0) {
                wait( );
        }
        spareResources--;
        // use resource
        spareResources++;
        notifyAll( );
}
```

static methods can be *synchronized*, using the lock from the *java.lang.Class* instance representing the *class.*

Thread Life Cycle



Important Methods in the *String class*

public char charAt (int index)
public String concat (String s)
public boolean equalsIgnoreCase (String s)
public int length( ) // be sure not to confuse with array length attribute
public String replace (char old, char new)
public String substring (int begin)
public String substring (int begin, int end) // begin is inclusive, end is exclusive (*0* to *length( )-1*)
public String toLowerCase( )
public String toString( )
public String toUpperCase( )
public String tim( )

The *StringBuffer class*

public string toString( )

The following methods return the altered StringBuffer and alter the instance calling the method.
public synchronized StringBuffer append(String s)
public synchronized StringBuffer insert(int offset, String s) // inserted after offset
public synchronized StringBuffer reverse( )

java.lang.Math class

public final static double Math.PI
public final static double Math.E

public static int abs(int input)
public static long abs(int long)
public static float abs(int float)
public static double abs(int double)

public static int max(int a, int b)
public static long max(long a, long b)
public static float max(float a, float b)
public static double max(double a, double b)

public static int min(int a, int b)
public static long min(long a, long b)
public static float min(float a, float b)
public static double min(double a, double b)

public static double random( ) // returns number in the range 0.0 to < 1.0

public static double ceil(double a)
public static double floor(double a)
public static int round(float a) // adds .5 then does a floor
public static long round(double a) // adds .5 then does a floor

public static double sin(double angrad)
public static double cos(double angrad)

public static double tan(double angrad)

public static double toRadian(double angdeg)
public static double toDegrees(double angdeg)

public static double sqrt(double a) // returns NaN when its argument is less than zero

NaN isn't == to anything not even itself instead use:
public boolean isNaN() in both *Float* and *Double*

Math.sqrt(Double.NEGATIVE_INFINITY) == Double.NaN

Primitive	Wrapper Class	Constructor
boolean	*Boolean*	*boolean* or *String*
byte	*Byte*	*byte* or *String*
char	*Character*	*char*
double	*Double*	*double* or *String*
float	*Float*	*float, double* or *String*
int	*Integer*	*int* or *String*
long	*Long*	*long* or *String*
short	*Short*	*short* or *String*

Common Wrapper Conversion Methods

xxxValue( )	returns primitive of Wrapper
static parseXxxx(String s)	returns primitive (*throws NumberFormatException*)
static parseXxxx(String s, int radix)	returns primitive in radix base (*throws* NFE)
static valueOf(String s)	returns Wrapper (*throws* NFE)
static valueOf(String s, int radix)	returns Wrapper in radix base (*throws* NFE)
toString()	returns String of Wrapper
static toString(primative)	returns String of primitive
static toString(primitive, int radix)	returns String of primitive in radix base
static toBinaryString( )	returns String of primitive in binary
static toHexString( )	returns String of primitive in hexadecimal
static toOctalString( )	returns String of primitive in octal

Method **s = static** **n = NFE**	**Boolean**	**Byte**	**Character**	**Double**	**Float**	**Integer**	**Long**	**Short**
byteValue		X		X	X	X	X	X
doubleValue		X		X	X	X	X	X
floatValue		X		X	X	X	X	X
intValue		X		X	X	X	X	X
longValue		X		X	X	X	X	X
shortValue		X		X	X	X	X	X
parseXxx s,n		X		X	X	X	X	X

parseXxx s,n (with radix)		X				X	X	X
valueOf s,n	X	X		X	X	X	X	X
valueOf s,n (with radix)		X				X	X	X
toString	X	X	X	X	X	X	X	X
toString s (primitive)		X		X	X	X	X	X
toString s (primitive, radix)						X	X	
toBinaryString s						X	X	
toHexString s						X	X	
toOctalString s						X	X	

Practice Question Notes

signed is not a valid modifier or keyword in Java

include in not a valid keyword, but *goto* is

single quotes are needed around character literal numbers i.e. *'\u0000'*

You can use *static* and *final* for a method declaration

Method visibility is a compile time error not a runtime error

interface variables are by default *final static*

interface variables cannot be *transient*

interface methods cannot be *static* or *final*

Implemented *interface* methods must always be *public* because access modifiers cannot restrict implemented methods more than their interface declaration

Widening casts happen implicitly

Constant expressions i.e. 3-1 are allowed as a case parameter but not variable expressions

Remember to check *boolean* statements for single equals =

Keywords can be used as identifiers if they start with a capital letter

Be careful of *while* loops nested in *do-while* loops.

Either a *catch* **or** *finally* must follow a *try* block.

If a method does not handle an exception, the *finally* block is executed first before the exception is propagated, otherwise the *catch* is executed first.

RuntimeException doesn't need to be declared.

The expression after the colon : in an *assert* statement must be able to be implicitly converted to a *String*.

It is sometimes good practice to *throw* an *AssertionError* explicitly.

Polymorphism uses the method of the instances' type and not the reference variable

```
class A {

        public void name( ) {
                System.out.println("A");
        }
}

class B extends A {

        public void name( ) {
                System.out.println("B");
        }
}

A a = new B( );
a.name( ); // prints B
```

Remember non-private variables are passed to subclasses.

Watch out for constructors with a *return* type.

(long) x/y only casts *x* and not *(x/y)* therefore

```
(long) x/y
≡
((long)x)/y
```

substring(int beginIndex, int endIndex)
beginIndex inclusive
endIndex exclusive
from *0* to *String.length( )-1*

Don't get *StringBuffer* and *String* methods confused.

Two primitive values of different types but equal in value will return *true* when tested for equality.

sqrt(double a) will return *Doubel.NaN* if *a* is negative.

xxxValue()	returns a primitive
parseXxx(String)	returns a primitive
valueOf(String)	returns a Wrapper

java.lang.StringBuffer uses the default *equals( )* and *hashCode( )* methods from *Object*

No duplicates are allowed in a *Set*

Watch for the *static* keyword on inner-classes

abstract types can be instantiated by using an anonymous *class* that overrides all *abstract* methods

Thread implements Runnable and so a *Thread* instance can be passed to the *Thread* constructor

The *sleep( )* methods *throws InterruptedException*

There are two versions of wait:

```
public final void wait( )
public final void wait(long timeout)   // on wait queue until notify( ), notifyAll( ) or
                                       // timeout ends
```

Thread has three *static* ints:

```
Thread.MIN_PRIORITY
Thread.NORM_PRIORITY
Thread.MAX_PRIORITY
```

Octal	Hexadecimal	Binary
1	1	1
8	16	2
64	256	4
512	4096	8
4096	65536	16

The *ClassLoader class* and the *Class class* can be used to load other classes

Method-local (local) inner classes are not associated with an instance of an outer (enclosing) *class.*

Valid identifiers begin with a Unicode letter, underscore character (_), or dollar sign ($). Subsequent characters consist of these characters and the digits 0-9.

The range of Char is 0 to 2^{16}-1

null is not a Java keyword.

Remember arithmetic rules are followed for Associativity therefore *, / and % have higher precedence than + and – which has higher precedence than boolean operators which have precedence in the following order &, ^, |, &&, ||. Otherwise expressions are left associative i.e. bracketed from left to right

<< left shift fills with zeros
>> right shift fills with sign bit (i.e. if negative stays negative)
>>> right shift fills with zeros

a >> b ≡ $a / 2^b$
a >>> b ≡ $| a / 2^b |$
a << b ≡ $a * 2^b$

A *float* or *double* value of Infinity cast to an integer is all-the-ones bit pattern
A *float* of *double* value of Negative Infinity cast to an integer is one-then-all-the-zeros bit pattern as follows:

Integer.toBinaryString((int)Float.NEGATIVE_INFINITY)

outputs

1000 0000 0000 0000 0000 0000 0000 0000

Integer.toBinaryString((int)Float.POSITIVE_INFINITY)

outputs

1111 1111 1111 1111 1111 1111 1111 1111

Java Operators, order of precedence - from highest to lowest

Priority	Operators	Operation	Associativity
	`[ ]`	array index	
1	`( )`	method call	left
	`.`	member access	
2	`++`	pre- or postfix increment	right
	`--`	pre- or postfix decrement	
	`+ -`	unary plus, minus	
	`~`	bitwise NOT	
	`!`	boolean (logical) NOT	
	`(type)`	type cast	

	`new`	object creation	
3	`* / %`	multiplication, division, remainder	left
4	`+ -`	addition, substraction	left
	`+`	string concatenation	
	`<<`	signed bit shift left	
5	`>>`	signed bit shift right	left
	`>>>`	unsigned bit shift right	
	`< <=`	less than, less than or equal to	
6	`> >=`	greater than, greater than or equal to	left
	`instanceof`	reference test	
7	`==`	equal to	left
	`!=`	not equal to	
8	`&`	bitwise AND	left
	`&`	boolean (logical) AND	
9	`^`	bitwise XOR	left
	`^`	boolean (logical) XOR	
10	`|`	bitwise OR	left
	`|`	boolean (logical) OR	
11	`&&`	boolean (logical) AND	left
12	`||`	boolean (logical) OR	left
13	`? :`	conditional	right
	`=`	assignment	
14	`*= /= += -= %= <<= >>= >>>= &= ^= |=`	combination assignment (operation and assignment)	right

References

Much of this text was copied from Sun Certified Programmer & Developer for Java 2 by Osborne Certification Press ISBN 0-07-222684-6.