OCA

Oracle Certified Associate

Java SE 8 Programmer I

STUDY GUIDE

SUMMARY

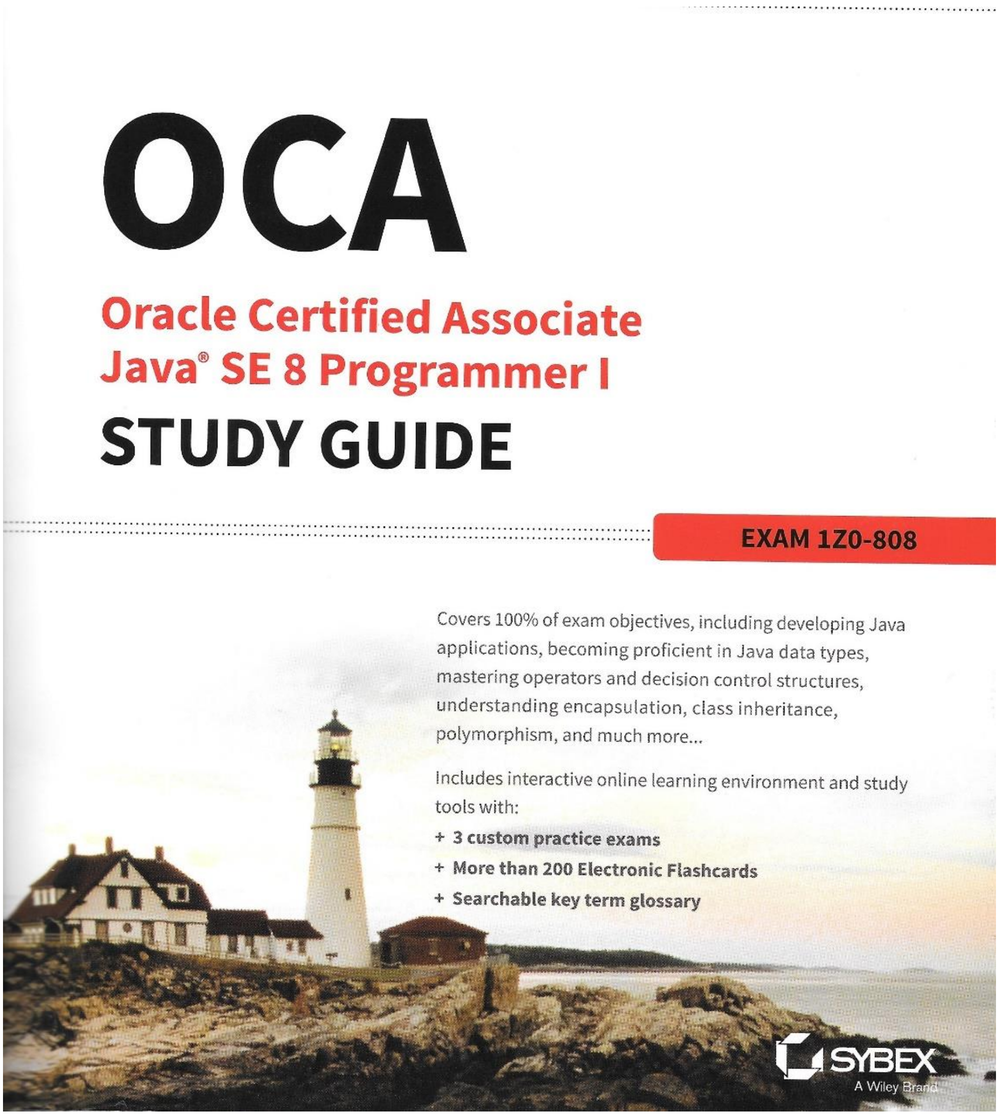


Table of Contents

Introduction

- URL blog: www.selikoff.net/oca → check for updates about changes on topics on the exam!
- Read Appendix B first.

1 Java Building Blocks

1.1 OCA Exam objectives

1. Java Basics:
   a. Define the scope of variables.
   b. Define the structure of a Java class.
   c. Create executable Java applications with a main method; run a Java program from the command line; including console output.
   d. Import other Java packages to make them accessible in your code.
   e. Compare and contrast the features and components of Java such as platform independence, object orientation, encapsulation, etc.
2. Working with Java Data Types:
   a. Declare and initialize variables (including casting of primitive types).
   b. Differentiate between object reference variables and primitive variables.
   c. Know how to read or write to object fields.
   d. Explain an Object's Lifecycle (creation, "dereference by reassignment" and garbage collection).

1.2 Understanding the Java Class Structure

- Classes are basic building blocks.
- A class describes all parts and characteristics of one of those building blocks.
- To use them you create objects.
- An object is a runtime instance of a class in memory.
- All various objects of all different classes represent the state of your program.

1.2.1 Fields and Methods

- Classes can have the following members:
  - fields (variables): hold the state of the program
  - methods (functions or procedures): operate on the state
- Keyword: a word with special meaning, like `public` and `class`, meaning a `public class` can be used by other classes.
- Example of a class with a field and methods:

```
public class Animal {         // defines a public class named Animal which can be used by other classes

   String name;               // defines a variable called name of type String

   public String getName() {  // defines public method that may be called from other classes
      return name;            // with return type String
   }

   public void setName(String newname) {  // defines public method which requires information to be
      name = newname;                     // supplied to it from calling method (parameter: newname
   }                                      // of type String and returns nothing (type void))
}
```

- The full declaration of a method is called a method signature.

1.2.2 Comments

- Comments aren't executable code, you can place them anywhere.
- Three types of comments:
  - Single-line comment:

```
// comment until end of line
```

- Multiple-line comment:

```
/* Multiple
 * line comment (the * here is optional, but added for readability)
 */
```

- Javadoc comment:

```
/**
 * Javadoc multiple-line comment
 * @author Jeanne and Scott
 */
```

1.2.3 Classes vs. Files

- Most of the time each Java class is defined in its own `*.java` file which is usually `public`, meaning any code can call it.
- You can put two classes in the same file, but at most one of the classes is then allowed to be `public`.
- If you have a `public class`, it needs to match the filename (`public class Animal2` would not compile in a file named `Animal.java`, but it would if the file is named `Animal2.java`).

1.3 Writing a *main()* Method

- A Java program begins execution with its `main()` method.
- Comments aren't executable code, you can place them anywhere.
- The signature is:

```
public class Zoo { // we need a class structure to start a Java program because the language requires it
   public static void main(String[] args) {     // this declares the entry point
   }
}
```

- To compile the source code we do: `javac Zoo.java`.
  - To compile it, it needs to have the extension `.java` and match the name of the class (including case).
  - The result is a file of bytecode with the same name but with the `.class` extension: `Zoo.class`.
  - Bytecode consists of instructions that the JVM knows how to execute.
- To execute the code we do: `java Zoo`.
  - We leave out the `.class` extensions on execution, since the period has a reserved meaning in the JVM.
- Main method signature:
  - ***public***
    - Access modifier → declares method's level of exposure to potential callers in the program; `public` means anywhere.
  - ***static***
    - Binds a method to its class so it can be called by just the class name, e.g. `Zoo.main()`. Java doesn't need to create an object to call the `main()` method.
    - If a `main()` method isn't present in the class we name with the `.java` executable, the process will throw an error and terminate. Even if a `main()` method is present, Java will throw an exception if it isn't static. A nonstatic `main()` method might as well be invisible from the point of view of the JVM.
  - ***void***
    - Represents the return type. In general it's good practice to use void for methods that change an object's state.
  - ***(String[] args)***
    - Represents the parameter list.
    - Can also be written as `String args[]` or `String… args`;
    - `args` hints this list contains values that were read in (arguments) when the JVM started (any name can be used though).
    - `[]` represents an array, a fixed-size list of items all of the same type.

        - … represents varargs (variable argument list).
        - Array indexes begin with 0.
        - Spaces are used to separate arguments on the command line, e.g. `java Zoo Bronx Zoo`:
            - `args[0]` → Bronx
            - `args[1]` → Zoo
        - For spaces inside an argument, surround them in quotes, e.g. `java Zoo "San Diego" Zoo`:
            - `args[0]` → San Diego
            - `args[1]` → Zoo
        - All command-line arguments are treated as String objects.
        - If you don't pass enough arguments and try to access it, Java prints out an exception, e.g. running `java Zoo Bronx` and then trying to access `args[1]` results in `java.lang.ArrayIndexOutOfBoundsException`.
- You need a JDK to compile because it includes a compiler.
- To run already compiled code, a JRE is enough.
- Java class files run on the JVM and therefore run on any machine with Java.

1.4 Understanding Package Declarations and Imports

- Java puts classes in packages, logical groupings for classes.
- In Java you need to tell it in which packages to look in to find code.
- An error like `Random cannot be resolved to a type` can mean two things:
    - A typo in the name of the class.
    - Omitting a needed import statement.
- Import statements tell Java in which packages to look in for classes.
- Java only looks for class names in the packages.
- Package names are hierarchical: start reading package name at the beginning.
- If package name begins with `java` or `javax`, it comes with the JDK.
- Detailed packages are called child packages: `com.amazon.java8book` is a child package of `com.amazon` (it's longer thus more specific).
- The rule for package names is that they are mostly letters or numbers separated by dots (the same as for variable names).
- The exam doesn't try to trick you by giving invalid package names, but it can trick you giving invalid variable names though.

1.4.1 Wildcards

- Classes in the same package are often imported together, e.g. `import java util.*;`
- The * is a wildcard that matches all classes in the package. It doesn't import child packages, fields, or methods; it imports only classes.
- Including so many classes doesn't slow down your program; the compiler figures out what's actually needed.

1.4.2 Redundant imports

- The package `java.lang` is automatically imported.
- You also don't have to import classes that are in the same package as the class importing it. Java automatically looks in the current package for other classes.
- Examples given that classes `Files` and `Paths` are located in `java.nio.file`:

```
import java.nio.file.*;        // CORRECT: import both at the same time
import java.nio.file.Files;    // CORRECT: import Files explicitly
import java.nio.file.Paths;    // CORRECT: import Paths explicitly
import java.nio.*;             // INCORRECT: * only matches class names, not subpackages as "file.*Files"
import java.nio.*.*;           // INCORRECT: only one wildcard allowed and must be at the end
import java.nio.file.Paths.*; // INCORRECT: cannot import methods, only class names
```

1.4.3 Naming Conflicts

- One of the reasons for using packages is so that names don't have to be unique across all of Java, e.g. you can have two different `Date` classes in different packages: `java.util.Date` and `java.sql.Date`.
- It can be tricky though when you have multiple imports like:

```
import java.util.*;
import java.sql.*;              // DOES NOT COMPILE: the type Date is ambiguous
```

But this will work:

```
import java.util.Date;          // explicitly imported class name has precedence over any wildcards present
import java.sql.*;
```

But with "ties" of precedence:

```
import java.util.Date;          // this will fail, there can't be two defaults; results in:
import java.sql.Date;           // The import java.sql.Date collides with another import statement
```

- If you need both classes, you can pick one to use in the import and for the other you use the fully qualified class name. Or you could leave out any import and for both use the fully qualified class name.

1.4.4 Creating a New Package

- If no package name is used in a class, it has the default package scope. This should not be used, only for throwaway code. In real life, always name your packages to avoid naming conflicts and to allow others to reuse your code.
- The directory structure on your computer is related to the package name.

1.4.5 Code Formatting on the Exam

- If the exam isn't asking about imports in the question, it will often omit the imports to save space. In that case you'll see examples with line numbers that don't begin with 1.
- When you do see the line number 1 or no line numbers at all, you have to make sure imports aren't missing.
- You'll also see code merged on the same line.
- You'll also see code that doesn't have a `main()` method. When this happens, assume the `main()` method, class definition, and all necessary imports are present.

1.5 Creating Objects

1.5.1 Constructors

- To create an instance of a class, you write `new` before it, e.g. `Random r = new Random();`
  - First you declare the type you'll be creating and then give the variable a name, i.e. a place to store a reference to the object.
  - Then you write `new Random()` to actually create the object.
- `Random()` is a constructor, a special type of method that creates a new object.
- Two key points of a constructor:
  - The name of the constructor matches the name of the class.
  - There is no return type.
- When you see a method name beginning with a capital letter and having a return type, pay special attention to it. It is not a constructor since there's a return type. It's a regular method.
- Purpose of constructor: initialize fields.
- Fields can also be initialized directly on the line on which they're declared, e.g. `int numEggs = 0;`
- If no constructor is defined, the compiler will supply a "do nothing" default constructor.

1.5.2 Reading and Writing Object Fields

- Reading a variable is known as getting it (getter).
- Writing to a variable is known as setting it (setter).

- You can read and write instance variables directly from the caller (depending on the access modifier used), e.g. `mother.numberEggs = 1` sets variable and `mother.numberEggs` reads variable.
- Fields can be read and written to directly on the line declaring them, e.g.
  `String first = "Theodore"; String last = "Moose"; String full = first + last;`
  first and last are written to and full reads first and last and writes to full.

1.5.3 Instance Initializer Blocks

- The code between braces – `{}` – is called a code block.
- Blocks can be inside a method and are run when the method is called.
- Blocks can be outside a method, called instance initializers.

1.5.3.1 Order of Initialization

- Fields and instance initializer blocks are run in the order in which they appear in the file.
- The constructor runs after all fields and instance initializer blocks have run
- You can't refer to a variable before it has been initialized.

1.6 Distinguishing Between Object References and Primitives

1.6.1 Primitive Types

- Java has eight built-in data types (primitive types).
- These represent the building blocks for Java objects; all Java objects are just a complex collection of these primitive data types.
- Java primitive types:

Keyword	Type	Example	Description
`boolean`	`true` or `false`	`true`	
`byte`	8-bit integral value	123	used for numbers without decimal points
`short`	16-bit integral value	123	used for numbers without decimal points
`int`	32-bit integral value	123	used for numbers without decimal points
`long`	64-bit integral value	123	used for numbers without decimal points
`float`	32-bit floating-point value	123.45f	used for floating-point (decimal) values requires letter f following the number
`double`	64-bit floating-point value	123.456	used for floating-point (decimal) values
`char`	16-bit Unicode value	'a'	

- Each numeric type uses twice as many bits as the smaller similar type.
- You should know that a byte can hold a value from –128 to 127.
- The number of bits is used by Java when it figures out how much memory to reserve for your variable, e.g. Java allocates 32 bits if you write `int num;`
- When a number is present in the code, it is called a literal.
- The following does not compile: `long max = 3123456789;` the value is seen as an `int` which can't be that large. To make it a long we have to do: `long max = 3123456789L;` by adding the `L` Java knows it's a long.
- Java allows you to specify digits in several other formats than base 10 (0 – 9):
  - Octal (digits 0–7), uses 0 as prefix, e.g. 017 which is base 8, so rightmost digit 7 is worth 7 and 2nd to rightmost digit is 1 is worth 8 (1 * 8), so in base 10: 7 + 8 = 15.
  - Hexadecimal (digits 0–9 and letters A–F), e.g. 0x1F which is base 16, so rightmost F is worth 15 and 2nd rightmost digit is 1 is worth 16 (1 * 16), so in base 10: 15 + 16 = 31.
  - Binary (digits 0–1), uses number 0 followed by a b or B as prefix, e.g. 0b11 which is base 2, so rightmost 1 is worth 1 and 2nd to rightmost digit 1 is worth 2 (1 * 2), so in base 10: 1 + 2 = 3.
- You won't need to convert between number systems on the exam though. You'll have to recognize valid literal values that can be assigned to numbers.

- Since Java 7 you can have underscores in numbers to make them easier to read as long as they are directly between two other numbers (so not at the beginning or the end of a literal and not right before or after a decimal point).

1.6.2 Reference Types

- A reference type refers to an object (instance of a class).
- Primitive types hold their values in memory where the variable is allocated.
- A reference "points" to an object by storing the memory address where the object is located (pointer).
- You can only use the reference to refer to the object.
- Examples:

```
java.util.Date date;  // date is a reference of type Date and can only point to a Date object
java.util.Date today; // today is also a reference of type Date and can only point to a Date object
String greeting;      // greeting is a reference that can only point to a String object
```

- A value is assigned to a reference in one of two ways:
  - A reference can be assigned to another object of the same type, e.g. `today = date();`
  - A reference can be assigned to a new object using the `new` keyword, e.g. `today = new java.util.Date();` or `greeting = "How are you?";`

1.6.3 Key Differences Between Primitives & Reference Types

- Reference types can be assigned `null`, meaning they do not currently refer to an object, primitives cannot be `null`; assigning `null` to a primitive will result in a compiler error.
- Reference types can be used to call methods when they do not point to `null`. Primitive types do not have methods declared on them.
- All primitive types have lower case type names. All classes that come with Java begin with uppercase.

1.7 Declaring and Initializing Variables

- A variable is a name for a piece of memory that stores data.
- When declaring a variable you state the variable type along with giving it a name, e.g. `String zooName;`
- Giving a variable a value is called initializing, which is done by using the equal sign after the variable name followed by the desired value, e.g. `zooName = "The Best Zoo";` or by doing it in the same statement as the declaration, e.g. `String zooName = "The Best Zoo";`

1.7.1 Declaring Multiple Variables

- You can declare many variables in the same declaration as long as they are all of the same type.
- You can also initialize any or all of those values inline.
- Examples:

```
String s1, s2;                // declared variables but not yet initialized

String s3 = "yes", s4 = "no"; // declared variables and initialized

int i1, i2, i3 = 0;           // tricky: only i3 is initialized, i1 and i2 are not yet initialized
                              // each snippet separated by a comma is a little declaration of its own

int num, String value;        // does not compile, tries to declare multiple variables of different
                              // types in the same statement; only works if they share the same type

double d1, double d2;         // not legal: not allowed to declare two different types in the same
                              // statement. Aren't they the same type (double)? Yes, but if you want
                              // to declare multiple variables in the same statement, they must share
                              // the same type declaration and not repeat it, thus this will work:

double d1, d2;                // is legal
```

1.7.2 Identifiers

- Three rules for valid identifiers:
  - The name must begin with a letter or the symbol $ or _.

- Subsequent characters may also be numbers.
- You cannot use the same name as a Java reserved keyword.

- List of Java reserved keywords (no need to memorize them for the exam, it will only ask about the ones you've already learned) – (* `const` and `goto` aren't actually used in Java):

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

- Consistency when declaring variable names is that it starts with a lowercase letter and then uses CamelCase, i.e. each word begins with an uppercase letter.
- Method and variable names begin with a lower case letter followed by CamelCase.
- Class names begin with an uppercase letter followed by CamelCase. Don't start any identifier with $ (although it is allowed), because the compiler uses this symbol for some files.

1.8 Understanding Default Initialization of Variables

1.8.1 Local Variables

- A local variable is a variable defined within a method.
- They must be initialized before use.
- They do not have a default value and contain garbage data until initialized.
- The compiler will not let you read an uninitialized value, e.g. `int y = 10; int x; int reply = x + y;` does not compile. Until `x` is assigned a value, it cannot appear within an expression.

1.8.2 Instance and Class Variables

- Variables that are not local variables are known as fields of the class, which can either be:
  - instance variables
  - class variables, which are shared across multiple objects and can be recognized by its keyword `static` in front of its name
- Instance and class variables do not require you to initialize them. As soon as they are declared they get a default value which is `null` for an object and `0`/`false` for a primitive:

Variable type	Default initialization value
`boolean`	`false`
`byte, short, int, long`	`0` (in the type's bit-length)
`float, double`	`0.0` (in the type's bit-length)
`char`	`'\u0000'` (NUL)
All object references (everything else)	`null`

1.9 Understanding Variable Scope

- Method parameters (variables passed into a method) are local to the method.
- Method parameters and variables declared inside a method have local scope to the method.
- Local variables can never have a scope larger than the method they are defined in.
- They can even have a smaller scope if they are declared within a block inside a method; each block of code, i.e. code between curly brackets – `{}` –, has its own scope. When there are multiple blocks, you match them from

the inside out. These smaller contained blocks can reference variables defined in the larger scoped blocks, but not vice versa.

- Instance variables are available as soon as they are defined and last for the entire lifetime of the object itself (until object is garbage collected).
- Class (static) variables go into scope when declared and stay in scope for the entire life of the program.

1.10 Ordering Elements in a Class

- Comments can go anywhere in the code.
- Other elements of a class:

Element	Example	Required?	Where does it go?
Package declaration	`package abc;`	No	First line in the file
Import statements	`import java.util.*;`	No	Immediately after the package
Class declaration	`public class C`	Yes	Immediately after the import
Field declarations	`int value;`	No	Anywhere inside a class
Method declarations	`void method()`	No	Anywhere inside a class

- Fields and methods must be within a class.
- Multiple classes can be defined in the same file, but only one of them is allowed to be `public`. The `public class` matches the name of the file.
- A file is also allowed to have neither class be `public`. As long as there isn't more than one `public class` in a file, it is okay.

1.11 Destroying Objects

Java provides a garbage collector to automatically look for objects, which are stored on the memory's heap, that aren't needed anymore.

1.11.1 Garbage Collection

- This is the process of automatically freeing memory on the heap by deleting objects that are no longer reachable in your program.
- You do need to know that `System.gc()` is not guaranteed to run; it suggests that now might be a good time for Java to kick off a garbage collection run. Java is free to ignore the request.
- You should be able to recognize when objects become eligible for garbage collection.
- Java waits patiently until the code no longer needs that memory. An object will remain on the heap until it is no longer reachable, which is when:
    - The object no longer has any references pointing to it.
    - All references to the object have gone out of scope.
- Objects vs. References:
    - Do not confuse a reference with the object that it refers to; they are two different entities.
    - The reference is a variable that has a name and can be used to access the contents of an object.
    - A reference can be assigned to another reference, passed to a method, or returned from a method.
    - An object sits on the heap and does not have a name, thus you can only access it through a reference.
    - An object cannot be assigned to another object, nor can an object be passed to a method or returned from a method.
    - It is the object that gets garbage collected, not its reference.

1.11.2 finalize()

- The `finalize()` method gets called if the garbage collector tries to collect the object.
- For the exam you need to know `finalize()` could run zero or one time: it might not get called and it definitely won't be called twice: if the garbage collector doesn't run `finalize()` will not be called, if the garbage collector fails, it will not be called a second time.

1.12 Benefits of Java

- ***Object oriented:*** all code is defined in classes and most of those can be instantiated into objects. Java allows for functional programming within a class, but object oriented is still the main organization of code.
- ***Encapsulation:*** Java supports access modifiers to protect data from unintended access and modification.
- ***Platform Independent:*** write once, run everywhere.
- ***Robust:*** prevents memory leaks (like in C++) since it manages memory on its own (automatic garbage collection).
- ***Simple:*** eliminates pointers and got rid of operator overloading (like in C++ `a + b` could have any meaning).
- ***Secure:*** runs inside JVM, which creates a kind of sandbox making it hard for Java code to do evil things to the computer it is running on.

1.13 Exam Essentials

- ***Be able to write code using a main() method:*** `public static void main(String[] args)`. Args are 0-indexed, so first argument is accessed by `args[0]`. Accessing an argument not passed in causes an exception.
- ***Understand the effect of using packages and imports:*** packages contain Java classes. Classes can be imported by class name or wildcard. Wildcards do not add subdirectories. In case of conflict, class name imports take precedence.
- ***Be able to recognize a constructor:*** has the same name as the class looking like a method but without a return type.
- ***Be able to identify legal and illegal declarations and initialization:*** multiple variables can be declared and initialized in the same statement when they share a type. Local variables require an explicit initialization; others use the default value for that type. Identifiers may contain letters, numbers, $, or _. Identifiers may not begin with numbers. Numeric literals may contain underscores between two digits and begin with 1-9, 0, 0x, 0X, 0b, and 0B.
- ***Be able to determine where variables go into and out of scope:*** all variables go into scope when they are declared. Local variables go out of scope when the block they are declared in ends. Instance variables go out of scope when the object is garbage collected. Class (static) variables remain in scope as long as the program runs.
- ***Be able to recognize misplaced statements in a class:*** package and import statements are optional, but if present both go before the class declaration in that order, i.e. first package name, then import statements, then class declaration (which is mandatory). Fields and methods are also optional and are allowed in any order within the class declaration.
- ***Know how to identify when an object is eligible for garbage collection:*** draw a diagram to keep track of reference and objects as you trace the code. When no arrows point to a box (object), it is eligible for garbage collection.

2 Operators and Statements

2.1 OCA Exam Objectives

1. Using Operators and Decision Constructs:
   a. Use Java operators; including parentheses to override operator precedence.
   b. Create if and if/else and ternary constructs.
   c. Use a switch statement.
2. Using Loop Constructs:
   a. Create and use while loops.
   b. Create and use for loops including the enhanced for loop.
   c. Create and use do/while loops.
   d. Compare loop constructs.
   e. Use break and continue.

2.2 Understanding Java Operators

- A Java operator is a symbol that can be applied to a set of variables, values, or literals (operands) that returns a result.
- Java knows unary, binary and ternary operators which can be applied to one, two, or three operands respectively.
- Java operators are not necessarily evaluated from left-to-right; unless overridden with parentheses, they follow order of operation (operator precedence) as follows (for OCA exam the ones in bold are relevant):

Operator	Symbols and examples
Post-unary operators	***expression++***, ***expression--***
Pre-unary operators	***++expression***, ***--expression***
Other unary operators	~, +, -, ***!***
Multiplication/Division/Modulus	*******, ***/***, ***%***
Addition/Subtraction	***+***, ***-***
Shift operators	<<, >>, >>>
Relational operators	***<***, ***>***, ***<=***, ***>=***, instanceof
Equal to/not equal to	***==***, ***!=***
Logical operators	***&***, ***^***, ***|***
Short-circuit logical operators	***&&***, ***||***
Ternary operators	***boolean expression ? expression1 : expression2***
Assignment operators	***=***, ***+=***, ***-=***, *=, /=, %=, &=, ^=, |=, <<=, >>=, >>>=

- Example: `int y = 4; double x = 3 + 2 * --y;`
  First y is decremented to 3, then multiplied by 2 resulting in 6, then added with 3 resulting in 9 upcasted to 9.0.

2.3 Working with Binary Arithmetic Operators

Binary operators (most common) perform mathematical operations on variables, create logical expressions and perform basic variable assignments. They are usually combined in complex expressions with more than two variables, thus operator precedence is very important in evaluating expressions.

2.3.1 Arithmetic Operators

- These include addition (`+`), subtraction (`–`), multiplication (`*`), division (`/`) and modulus (`%`).
- They also include unary operators `++` and `––`.
- Multiplicative operators (`*`, `/`, `%`) have higher order of precedence than additive operators (`+`, `–`):
  `int x = 2 * 5 + 3 * 4 – 8;` → `int x = 10 + 12 – 8;` → remaining terms are evaluated in left-to-right order resulting in `14`.
- Order of operation can be changed by using parentheses around the sections you want to evaluate first:

`int x = 2 * ((5 + 3) * 4 – 8);` → `int x = 2 * (8 * 4 – 8);` → `int x = 2 * (32 – 8);` → `int x = 2 * 24` resulting in `48` for `x`.

- All of the arithmetic operators may be applied to any Java primitives, except boolean.
- Only `+` and `+=` may be applied to `String` values (`String` concatenation).
- The modulus (remainder) operator is the remainder when two numbers are divided, e.g. `9 / 3` divides evenly into `3`, thus `9 % 3 = 0`; but `11 / 3` doesn't divide evenly, thus `11 % 3 = 2`.
- For integer values, division results in the floor value of the nearest integer that fulfills the operation, whereas modulus is the remainder value.
- For a given divisor `y`, the modulus operation results in a value between `0` and `(y – 1)` for positive dividends.
- The modulus operation may also be applied to negative integers and floating-point numbers; for a given divisor `y` and negative dividend, the resulting modulus value is between (`–y + 1`) and `0` (this is not relevant for the exam).

2.3.2 Numeric Promotion

- If two values have different data types, Java will automatically promote one of the values to the larger of the two data types.
- If one of the values is integral and the other is floating-point, Java will automatically promote the integral value to the floating-point value's data type.
- Smaller data types (`byte`, `short`, and `char`) are first promoted to `int` any time they're used with a Java binary arithmetic operator, even if neither of the operands is `int` (not the case for unary operator).
- After all promotion has occurred and the operands have the same data type, the resulting value will have the same data type as its promoted operands.
- Examples:

```
int x = 1; long y = 33; x * y;              // result data type is a long
double x = 39.21; float y = 2.1; x + y;     // does not compile: should be float y = 2.1f; then
                                            // result data type would be double
short x = 10; short y = 3; x / y;           // x and y will be promoted to int before the operation
short x = 14; float y = 13; double z = 30;  // all rules apply: x promotes to int, then to float so
  x * y / z;                                // it can multiplied by y (in this case it doesn't have
                                            // to be 13f), x * y promotes to double so that it can be
                                            // divided with z, resulting in datatype double
```

2.4 Working with Unary Operators

A unary operator requires exactly one operand, or variable to function, e.g. for increasing a numeric variable by one, or negating a boolean variable.

Unary operator	Description
`+`	Indicates a number is positive, which is the default unless accompanied by a negative unary operator.
`–`	Indicates a literal number is negative or negates an expression.
`++`	Increments a value by 1.
`--`	Decrements a value by 1.
`!`	Inverts a boolean's logical value.

2.4.1 Logical Complement and Negation Operators

- The logical complement operator (`!`) flips the value of a boolean expression: true becomes false, false becomes true.
- The negation operator (–) reverses the sign of a numeric expression.
- Some operators require the variable or expression they're acting upon to be of a specific type: the negation operator cannot be applied to a boolean expression (false is not equal to 0 and true is not equal to 1) and a logical complement operator cannot be applied to a numeric expression. **Watch out for this on the exam!**

2.5 Increment and Decrement Operators

- Increment (++) and decrement (−−) operators can be applied to numeric operands and have the higher order or precedence, as compared to binary operators.
- They often get applied to an expression.
- Pre-increment and pre-decrement operator: the operator is placed before the operand; the operator is applied first and the value returned is the new value of the expression.
- Post-increment and post-decrement operator: the operator is placed after the operand; the original value of the expression is returned and the operator is applied after the value is returned.
- **Watch out on the exam:** multiple increment or decrement operators can be applied to a single variable on the same line, e.g.

```
int x = 3; int y = ++x * 5 / x-- + --x;
int y = 4 * 5 / x-- + --x;              // x assigned value of 4
int y = 4 * 5 / 4 + --x;                // x assigned value of 3
int y = 4 * 5 / 4 + 2;                  // x assigned value of 2; 4 * 5 = 20 / 4 = 5 + 2 = 7

Result is: x is 2 and y is 7.
```

2.6 Using Additional Binary Operators

- An assignment operator is a binary operator that modifies, or assigns, the variable on the left-hand side of the operator, with the result of the value on the right-hand side of the equation, e.g.: `int x = 1;` assigns 1 to x.
- Java automatically promotes from smaller to larger data types; it throws a compiler exception if it detects you are trying to convert from larger to smaller data types.

2.6.1 Casting Primitive Values

- We can assign a larger numerical data type to a smaller numerical data type by applying casting or by converting from a floating-point number to an integral value, e.g.:

```
int x ((int)1.0;
short y = (short)1921222;        // too large to store (numeric overflow); stored as 20678
int z = (int)9f;
long t = 192301398193810323L;
```

- Overflow is when a number is so large that it will no longer fit within the data type; the number “wraps around” to the next lowest value and counts up from there.
- When a number is too low to fit in the data type, we speak of underflow (out of scope for the exam).
- The following does not compile:

```
short x = 10;
short y = 3;
short z = x * y;       // DOES NOT COMPILE because x and y are promoted to int causing z to be an int.
```

To make it compile, we need to cast the result of the multiplication to a short:

```
short z = (short)(x * y);        // tells the compiler to ignore its default behavior and you need to
                                 // take care to prevent overflow or underflow
```

2.6.2 Compound Assignment Operators

- Only two for the OCA exam are required to know: += and −=.
- Examples:

```
int x = 2, z = 3;
x = x * z;             // simple assignment operator
x *= z;                // compound assignment operator: result is the same as with the simple assignment
                       // operator (it's a shorthand notation)
```

- The left-hand side of the compound operator can only be applied to a variable that is already defined and cannot be used to declare a new variable.
- The shorthand notation can save us from having to explicitly cast a value:

```
long x = 10; int y = 5;
y = y * x;              // DOES NOT COMPILE
y = (int)(y * x);       // DOES COMPILE
y *= x;                 // DOES COMPILE: compound operator will first cast y to long, apply multiplication
                        // of two long values, and then cast the result to an int
```

- The result of the assignment operator is an expression in and of itself, equal to the value of the assignment:

```
long x = 5; long y = (x = 3);
System.out.println(x);    // outputs 3
System.out.println(y);    // also outputs 3: sets value of variable x to 3, then returns value of
                          // assignment, which is also 3
```

2.6.3 Relational Operators

- Relational operators compare two expressions and return a boolean value.
- Relational operators are (for the examples: `int x = 10, y = 20, z = 10;`):

Relational operator	Description	Example	Result
`<`	Strictly less than.	`x < y`	true
`<=`	Less than or equal to.	`x <= y`	true
`>`	Strictly greater than.	`x >= z`	true
`>=`	Greater than or equal to.	`x > z`	false
`a instanceof b`	True if reference that a points to is an instance of a class, subclass, or class that implements a particular interface, as named in b. Out of scope for OCA exam.		

- The first four from above table are applied to numeric primitive data types; if the two numeric operands are not of the same data type, the smaller one is promoted as discussed before.
- The instanceof is applied to object references and classes or interfaces.

2.6.4 Logical Operators

- Logical operators, (`&`), (`|`) and (`^`), may be applied to both numeric and boolean data types.
- Called logical operators when applied to boolean data types.
- Called bitwise operators when applied to numeric data types. For the OCA exam you don't need to know anything about numeric bitwise comparison.
- Logical truth tables for `&`, `|`, and `^`:

x & y (AND)

	y = true	y = false
x = true	true	false
x = false	false	false

x | y (INCLUSIVE OR)

	y = true	y = false
x = true	true	true
x = false	true	false

x ^ y (EXCLUSIVE OR)

	y = true	y = false
x = true	false	true
x = false	true	false

  - And is only true if both operands are true.
  - Inclusive OR is only false if both operands are false.
  - Exclusive OR is only true if the operands are different.
- Conditional operators, (`&&`) and (`||`) , are also called short-circuit operators: right-hand side of the expression may never be evaluated if the final result can be determined by left-hand side of the expression, e.g.:

```
if (x != null && x.getValue() < 5) { … }        // if x is null, x.getValue() < 5 is never evaluated
if (x != null & x.getValue() < 5) { … }         // both will be evaluated, so if x is null, x.getValue()
                                                // will throw a NullPointerException
```

- **Be wary** of short-circuit behavior on the exam, as questions are known to alter a variable on the right-hand side of the expression that may never be reached.

2.6.4.1 Equality Operators

- There is a semantic difference between "two objects are the same" and "two objects are equivalent".
- For numeric and boolean primitives, there is no such distinction.

- The equals operator (==) and not equals operator (!=) compare two operands and return a boolean value whether the expressions or values are equal, or not equal, respectively.
- Equality operators can be used in three scenarios:
    - Comparing two numeric primitive types: if numeric values are of different data types, the values are automatically promoted as described earlier, e.g. `5 == 5.00` returns `true`.
    - Comparing two boolean values.
    - Comparing two objects, including `null` and `String` values.
- You cannot mix and match types, e.g.: `boolean x = true == 3;` does not compile.
- **Pay close attention** to the data types when you see an equality operator on the exam; assignment operators and equality operators are often mixed, e.g.:

```
boolean y = false; boolean x = (y = true);
System.out.prinln(x);                    // outputs true
```

- For object comparison, the equality operator is applied to the references to the objects, not to the objects they point to. Two references are equal if and only if they point to the same object, or both point to null, e.g.:

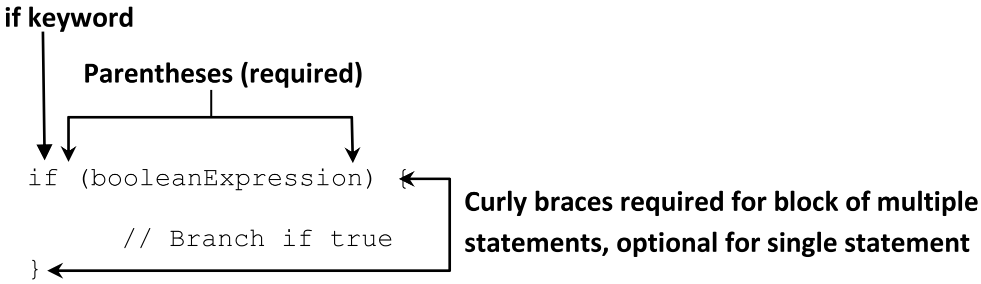
```
File x = new File("a.txt"); File y = new File("a.txt"); File z = x;
System.out.println(x == y);              // outputs false
System.out.println(x == z);              // outputs true
```

2.7 Understanding Java Statements

- A Java statement is a complete unit of execution in Java, terminated with a semicolon (;).
- Control flow statements break up the flow of execution by using decision making, looping, and branching, allowing the application to selectively execute particular segments of code.
- Statements can be applied to single expressions as well as a block of Java code, the latter being a group of zero or more statements between balanced braces ({}) which can be used anywhere a single statement is allowed.

2.7.1 The if-then Statement

- We use the `if-then` statement if we only want to execute a block of code under certain circumstances, i.e. it will be executed if and only if a boolean expression evaluates to `true` at runtime.
- The structure of an `if-then` statement:

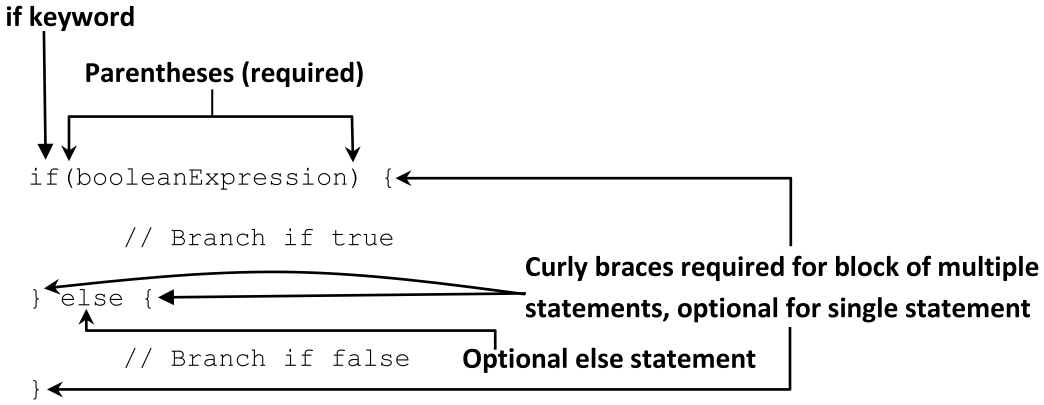


- A statement block allows multiple statements to be executed based on the `if-then` evaluation. For readability it is good practice to put blocks around the execution component of `if-then` statements (even if it is only one statement).
- **On the exam watch out for if-then statements without braces ({}).** Be sure to trace the open and close braces of the block and ignore any indentation you may come across.

2.7.2 The if-then-else Statement

- To branch between one of two possible options, we could write two different statements, but a better way is to use the `if-then-else` statement in which the boolean evaluation is done only once.
- The `else` operator takes a statement or block of statements.

- The structure of an `if-then-else` statement:



- We can append additional `if-then` statements to an `else` block. The Java process will continue execution until it encounters an `if-then` statement that evaluates `true` and if none found it will execute the final code of the `else` block, e.g.:

```
if (…) {
   // statements
} else if (…) {
   // statements
} else {
   // statements
}
```

- Note though that in the above example of a nested `if-then-else`, order is important. You need to be careful not to create unreachable code, e.g.:

```
if (hourOfDay < 15) {
   // statements
} else if (hourOfDay < 11) {
   // statements      // UNREACHABLE CODE: only one branch can be executed here - if the first branch is
                      // executed, then the second cannot be executed
} else {
   // statements
}
```

- **On the exam watch out for code where the boolean expression inside the `if-then` statement is not actually a boolean expression:** `int x = 1; if (x) { … }` does not compile.
- **Also be wary of assignment operators being used as if they were `equals` (==) operators in `if-then` statements:** `int x = 1; if (x = 5) { … }` does not compile.

2.7.3 Ternary Operator

- The ternary operator, or conditional operator, `? :`, takes three operands in the form of:

$$\texttt{booleanExpression ? expression}_1\texttt{ : expression}_2$$

- The first operand must be a boolean expression, the second and third can be any expression that returns a value.
- It is a condensed form of an `if-then-else` statement that returns a value.
- The second and third expressions don't have to be of the same data types, but it may come into play when combined with the assignment operator:

```
System.out.println((y > 5) ? 21 : "Zebra";   // COMPILES: the method System.out.println can convert
                                             // both expressions to String
int animal = (y < 91) ? 9 : "Horse";         // DOES NOT COMPILE: String Horse cannot be assigned to int
```

- Only one of the right-hand expressions of the ternary operator will be evaluated at runtime. Like short-circuit operators, if one of the two right-hand expressions in a ternary operator performs a side effect, then it may not be applied at runtime, in other words: the expressions in a ternary operator may not be applied if the particular expression is not used. **At the exam, be wary of any question that includes a ternary expression in which a variable is modified in one of the right-hand side expressions:** e.g.:

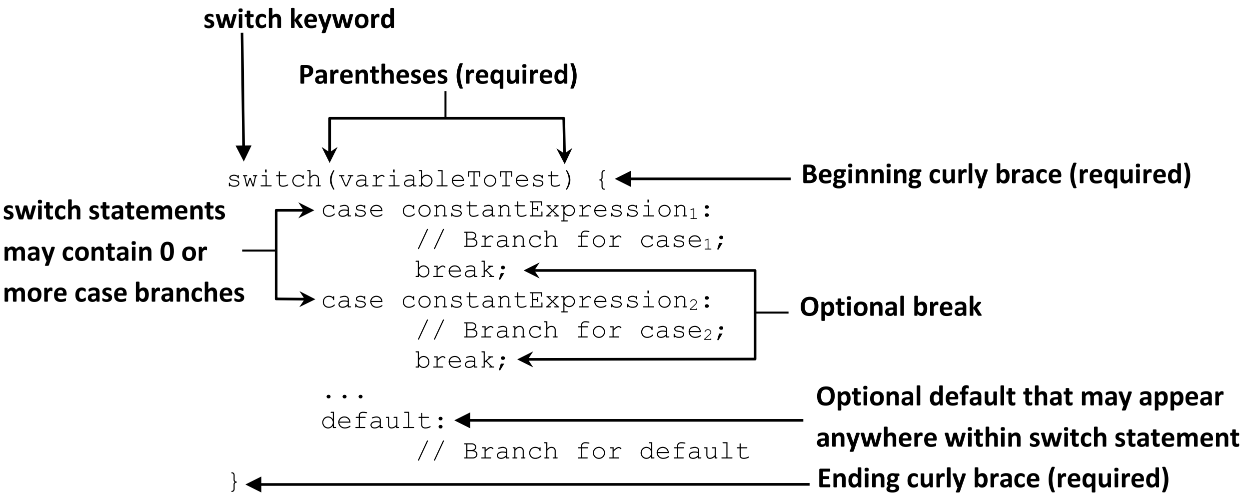
```
int y = 1; int z = 1;
final int x = y<10 ? y++ : z++;     // left-hand boolean expression is true, so only y is incremented
                                     // resulting in y having value of 2 and z value of 1
int y = 1; int z = 1;
final int x = y>=10 ? y++ : z++;    // left-hand boolean evaluates to false, so only z is incremented
                                     // resulting in y having value of 1 and z value of 2
```

2.7.4 The switch Statement

- The switch-statement is a complex decision-making structure in which a single value is evaluated and flow is redirected to the first matching branch (case statement).
- If a case statement is found that matches the value, an optional `default` statement will be called.
- If no such `default` option is available, the entire `switch` statement will be skipped.

2.7.4.1 Supported Data Types

- A `switch` statement has a target variable that is not evaluated until runtime.
- Data types supported by switch statements are:
  - byte and Byte
  - short and Short
  - char and Character
  - int and Integer
  - String
  - enum values
- The structure of a `switch` statement:



2.7.4.2 Compile-time Constant Values

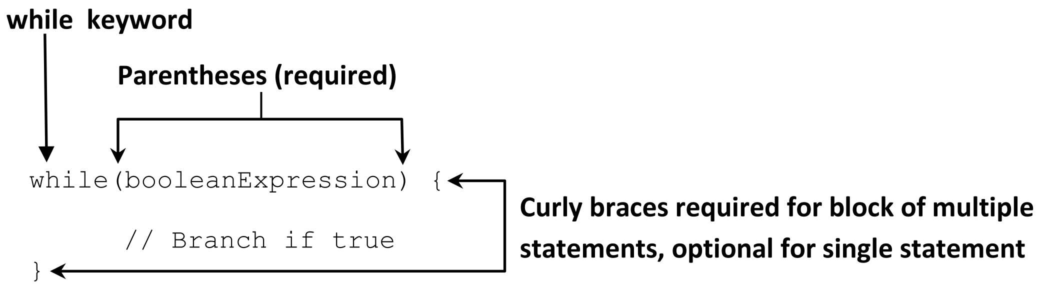
- The values in each `case` statement must be compile-time constant values of the same data type as the `switch` value, thus only literals, `enum` constants, or `final` constant variables of the same data type can be used.
- A `final` constant is marked with the `final` modifier and initialized with a literal value in the same expression in which it is declared.
- A `break` statement at the end of a `case` or `default` statement will terminate the `switch` statement and return flow control to the enclosing statement.
- If you leave out the `break` statement, flow will continue to the next proceeding `case` or `default` block automatically.
- The `case` or `default` statements don't have to be in any particular order, unless you are going to have pathways that reach multiple sections of the `switch` block in a single execution.
- The `default` block is only branched to if there is no matching `case` value for the `switch` statement, regardless of its position within the `switch` statement.

- While the code will not branch to the `default` statement if there is a matching `case` value within the `switch` statement, it will execute the `default` statement if it encounters it after a `case` statement for which there is no terminating `break` statement (**in this case order of the `default` and `case` statements does matter!**).
- **The exam creators are fond of `switch` examples that are missing `break` statements!** Always consider that multiple branches may be visited in a single execution.
- The data type for `case` statements must all match the data type of the `switch` variable.
- Examples:

```
private int getSortOrder(String firstName, final String lastName) {
   String middleName = “Patricia”;
   final String suffix = “JR”;
   int id = 0;
   switch(firstName) {
      case “Test”:        // COMPILES: using String literal
         return 52;       // return statement also exits the switch (like a break)
      case middleName:    // DOES NOT COMPILE: middleName is not final
         id = 5; break;
      case suffix:        // COMPILES: suffix is a final constant variable
         id = 0; break;
      case lastName:      // DOES NOT COMPILE: is final, but not a constant (passed into function)
         id = 8; break;
      case 5:             // DOES NOT COMPILE: String does not match with data type of switch variable
         id = 7; break;
      case ‘J’:           // DOES NOT COMPILE: char does not match with data type of switch variable
         id = 10; break;
      case java.time.DayOfWeek.SUNDAY:  // DOES NOT COMPILE: enum does not match with data type of
         id = 15; break;                //                    switch variable
   }
   return id;
}
```

2.7.5 The while Statement

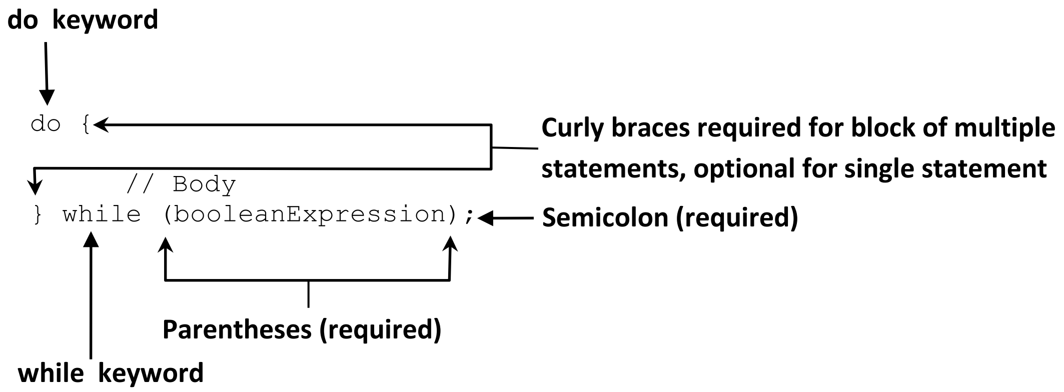
- A repetition control structure (loop) executes a statement of code multiple times in succession.
- By using nonconstant variables, each repetition of the statement may be different.
- The `while` statement is the simplest repetition control structure.
- It has a termination condition, implemented as a boolean expression, which continues as long as the expression evaluates to `true`.
- The structure of a `while` statement:



- During execution, the boolean expression is evaluated before each iteration of the loop and exits if the evaluation returns `false`.
- A `while` loop may terminate after its first evaluation of the boolean expression, meaning the statement block may never be executed.
- The termination condition of a `while` statement can consist of a compound boolean statement, e.g.: `while (x > 0 && y > 0) { … }`.
- Be careful not to create an infinite loop, like: `int x = 2, int y = 5; while(x<10) y++;`
  You should be absolutely certain that the loop will eventually terminate under some condition: make sure the loop variable is modified, then ensure that the termination condition will be eventually reached in all circumstances.

2.7.6 The do-while Statement

- The `do-while` loop, like the `while` loop, also has a termination condition and statement, or block of statements, but it is guaranteed to be executed at least once, because it first executes the statement(s) and then checks the loop condition.
- The structure of a `do-while` statement:



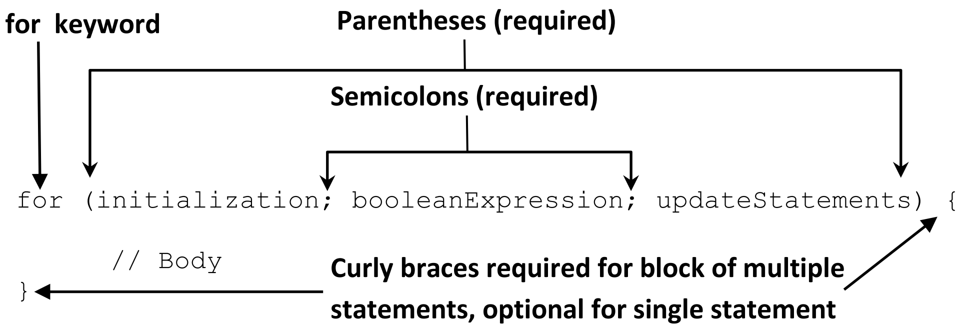
- Java recommends you use a `while` loop when a loop might not be executed at all and a `do-while` loop when the loop is executed at least once.

2.7.7 The for Statement

There are two variations of the `for` statement: the basic `for` loop and the enhanced `for` loop (`for-each` statement).

2.7.7.1 The Basic for Statement

- A basic `for` loop has the same conditional boolean expression and statement, or block of statements, as the other loops. Additionally, it also has an initialization block and an update statement.
- The structure of a basic `for` statement:



1. Initialization statement executes.
2. If booleanExpression is `true` continue, else exit loop.
3. Body executes.
4. Execute updateStatements.
5. Return to Step 2.

- Each section is separated by a semicolon. The initialization and update sections may contain multiple statements, separated by commas.
- Variables declared in the initialization block of a `for` loop have limited scope and are only accessible within the `for` loop. **Watch out for this on the exam!**
- Variables declared before the `for` loop and assigned a value in the initialization block may be used outside the `for` loop because their scope precedes the `for` loop.
- The boolean condition is evaluated on every iteration of the loop before the loop executes.
- The components of the `for` loop are each optional. Leaving them all out will create an infinite loop. The semicolons separating the three sections are required though, e.g.:

```
for ( ; ; ) { System.out.println("Hello World"); }     // valid for loop, but infinite loop
for ( ) { … }                                          // will not compile
```

- Multiple terms can be added to the `for` statement:

```
int x = 0;                       // variable x can be declared before the loop begins
for (long y = 0, z = 4;          // init/boolean expression/update statements can include extra variables
     x < 5 && y < 10;            //   that may not reference each other: z is initialized but never used
     x++, y++) {                 // update statement can modify multiple variables
   System.out.print(y + " ");    // prints: 0 1 2 3 4
}
System.out.println(x);           // x can be used after the loop ended (declared outside loop) (prints: 5)
```

- It's not possible to redeclare a variable in the initialization block (this results in a compilation error):

```
int x = 0;
for (long y = 0, x = 4;            // DOES NOT COMPILE: x is already declared before the loop and repeated
     x < 5 && y < 10;              // in the initialization block > duplicate variable declaration
     x++, y++) {
}

int x = 0; long y = 10;
for (y = 0, x = 4;                 // DOES COMPILE: init assigns value to x but does not declare it
     x < 5 && y <10;
     x++, y++) {
}
```

- It's not possible to use incompatible data types in the initialization block:

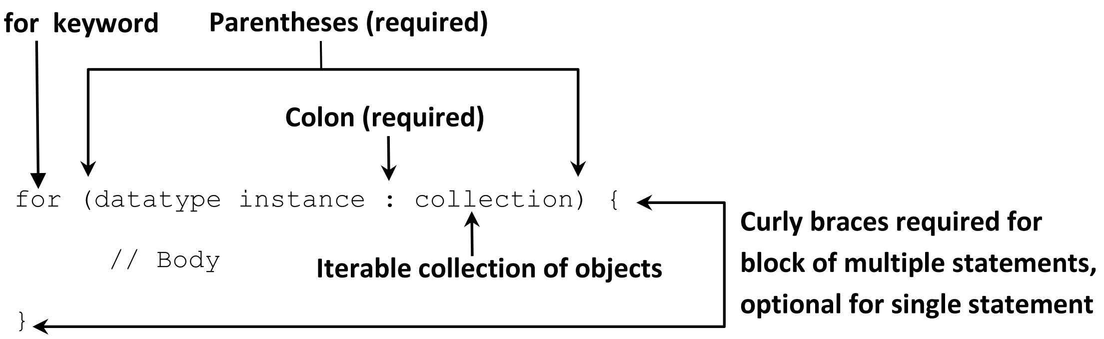
```
for (long y = 0, int x = 4;        // DOES NOT COMPILE: variables in initialization block must all be of
     x < 5 && y < 10;              // the same type, so either both y and x should be long or int
     x++, y++) {
}
```

- It's not possible to use loop variables outside the loop:

```
for (long y = 0, x = 4; x < 5 && y < 10; x++, y++) { … }
System.out.println(x);           // DOES NOT COMPILE
```

2.7.7.2 The for-each Statement

- The `for-each` loop is specifically designed for iterating over arrays and `Collection` objects.
- The structure of an enhanced `for-each` statement:



- The `for-each` declaration consists of an initialization section and an object to loop over.
- The right-hand side of the `for-each` loop statement must be a built-in Java array or an object whose class implements `java.lang.Iterable` (most of the Java `Collections` framework).
- The left-hand side must include a declaration for an instance of a variable, whose type matches the type of a member of the array or collection in the right-hand side of the statement.
- On each iteration of the loop, the named variable on the left-hand side of the statement is assigned a new value from the array or collection on the right-hand side of the statement; the datatypes need to be the same, e.g.:

```
String[] names = new String[3];
for (int name : names) { … }     // DOES NOT COMPILE: names is an array of String therefore the named
                                 // variable needs to be a String also and not an int
```

- **For the OCA exam, the only members of the `Collections` framework you need to know are `List` and `ArrayList`.**
- When you see a `for-each` loop on the exam, make sure the right-hand side is an array or `Iterable` object and the left-hand side has a matching type, e.g.:

```
String names = "Lisa";
for (String name : names) { … } // DOES NOT COMPILE: names is not an array / doesn't implement Iterable
```

- The `for-each` loop is convenient for working with lists, but it hides access to the loop iterator variable. For example if we wanted to print something only on the first occurrence of the loop, we would usually use a basic `for` loop where we have access to the iterator variable.
- It is also common to use a standard `for` loop over a `for-each` loop if comparing multiple elements in a loop within a single iteration, e.g. `values[i] - values[i-1]`.

2.8 Understanding Advanced Flow Control

2.8.1 Nested Loops

- Loops can contain other loops. For example, we can iterate over a two-dimensional array, an array that contains another array as its members.
- Nested loops can include `while` and `do-while`.

2.8.2 Adding Optional Labels

- `If-then` statements, `switch` statements and `loops` can all have optional labels.
- A label is an optional pointer to the head of a statement that allows the application flow to jump to it or break from it. It is a single word proceeded by a colon (`:`) and it's often used in loop structures.
- When dealing with only one loop, a label doesn't add any value. But they are useful when using nested loops.
- It is possible to add optional labels to control and block structures (this is not on the OCA exam and it's not good coding practice).
- Labels are usually in uppercase, with underscores between words (to distinguish them from regular variables).

2.8.3 The break Statement

- A `break` statement transfers the flow of control out to the enclosing statement. The same goes for `break` statements appearing inside of `while`, `do-while`, and `for` loops, as it will end the loop early.
- The structure of a `break` statement:

Optional reference to head of loop

Colon (required if optionalLabel is present)

```
optionalLabel: while (booleanExpression) {
       // Body
       // Somewhere in loop
       break optionalLabel;
}
```

Semicolon (required)

break keyword

- The `break` statement can take an optional label parameter. Without a label parameter, the `break` statement will terminate the nearest loop it is currently in the process of executing.
- The optional label parameter allows us to break out of a higher level outer loop.

2.8.4 The continue Statement

- The `continue` statement causes the flow to finish the execution of the current loop.

- The structure of a `continue` statement:

Optional reference to head of loop

Colon (required if optionalLabel is present)

```
optionalLabel: while (booleanExpression) {
      // Body
      // Somewhere in loop
      continue optionalLabel;
}
```

Semicolon (required)

continue keyword

- The `break` statement transfers control to the enclosing statement, the `continue` statement transfers control to the boolean expression that determines if the loop should continue, in other words: it ends the current iteration of the loop.
- The `continue` statement, like the `break` statement, is applied to the nearest inner loop under execution using optional label statements to override this behavior.
- Advanced flow control usage table:

	Allows optional labels	Allows unlabeled break	Allows continue statement
`if`	Yes [*]	No	No
`while`	Yes	Yes	Yes
`do while`	Yes	Yes	Yes
`for`	Yes	Yes	Yes
`switch`	Yes	Yes	No

[*] Labels are allowed for any block statement, including those that are preceded with `if-then` statements.

2.9 Exam Essentials

- ***Be able to write code that uses Java operators.***
- ***Be able to recognize which operators are associated with which data types:*** some are only applied to numeric primitives, some only to boolean values, and some only to objects. It's important to notice when operator and operand(s) mismatch.
- ***Understand Java operator precedence.***
- ***Be able to write code that uses parentheses to override operator precedence.***
- ***Understand if and switch decision control statements:*** these often appear in exam questions unrelated to decision control.
- ***Understand loop statements.***
- ***Understand how break and continue can change flow control.***

3 Core Java APIs

3.1 OCA Exam Objectives

1. Using Operators and Decision Constructs:
   a. Test equality between Strings and other objects using == and `equals()`.
2. Creating and Using Arrays:
   a. Declare, instantiate, initialize and use a one-dimensional array.
   b. Declare, instantiate, initialize and use a multi-dimensional array.
3. Working with Selected classes from the Java API:
   a. Creating and manipulating Strings.
   b. Manipulate data using the `StringBuilder` class and its methods.
   c. Declare and use an `ArrayList` of a given type.
   d. Create and manipulate calendar data using classes from `java.time.LocalDateTime`, `java.time.LocalDate`, `java.time.LocalTime`, `java.time.format.DateTimeFormatter`, `java.time.Period`.
4. Working with Java Data Types:
   a. Develop code that uses wrapper classes such as `Boolean`, `Double` and `Integer`.

Note: API (application programming interface) can be a group of class or interface definitions that gives you access to a service or functionality.

3.2 Creating and Manipulating Strings

A string is basically a sequence of characters, which can be created in two ways:

```
String name = "Fluffy";             // reference variable name pointing to String object "Fluffy",
                                    // not to instantiate with new → lives in the String Pool
String name = new String("Fluffy"); // also a reference variable name pointing to String object
                                    // "Fluffy"
```

3.2.1 Concatenation

- Strings can be concatenated by using the + operator, called string concatenation, e.g. `"1" + "2"` → `"12"`.
- The following rules count for string concatenation:
  1. If both operands are numeric, + means numeric addition.
  2. If either operand is a `String`, + means concatenation.
  3. The expression is evaluated left to right.

```
System.out.println(1 + 2);          // 3 (rule 1 applied)
System.out.println("a" + "b");      // "ab" (rule 2 applied)
System.out.println("a" + "b" + 3);  // "ab3" (rules 2 & 3 applied in that order)
System.out.println(1 + 2 + "c");    // "3c" (rules 3, 1 & 2 applied in that order)
```

- **Exam will trick you with things like (when seeing this, take your time and check the types):**

```
int three = 3;
String four = "4";
System.out.println(1 + 2 + three + four);       // "64" (rules 3, 1, 1 and 2 applied in that order)
```

- `s += "2"` means the same as `s = s + "2";`
- Recap: use numeric addition if two numbers are involved, use concatenation otherwise, and evaluate from left to right.

3.2.2 Immutability

- Once a `String` object is created, it is not allowed to change. It cannot be made larger or smaller, you cannot change one of the characters inside it.
- Mutable is another word for changeable. Immutable is the opposite: an object that can't be changed once it's created.
- On the OCA exam, you need to know that `String` is immutable.

- Immutable only has a getter. Mutable has a setter as well to allow the reference to change to point to a different `String`.
- Immutable classes are final, so subclasses can't add mutable behavior.
- Example:

```
String s1 = "1";
String s2 = s1.concat("2");
s2.concat("3");
System.out.println(s2);          // prints "12"; note that the 3rd statement is not assigned to a reference
                                 // variable!
```

3.2.3 The String Pool

- Strings use up a lot of memory, therefore in Java common strings can be reused. This is where the *string pool* (intern pool) comes in, a location in the JVM that collects all common strings.
- The string pool contains literal values like "name", but `myObject.toString()` is a `String` but not a literal and therefore doesn't go into the string pool.
- Strings not in the string pool are garbage collected just like any other object.

3.2.4 Important String Methods

- A string is a sequence of characters and Java counts from 0 when indexed:

a	n	i	m	a	l	s
0	1	2	3	4	5	6

- You need to know how to use the following string methods below.

3.2.4.1 length()

- Returns the number of characters in the `String`.
- Signature: `int length()`
- Example:

```
String string = "animals";
System.out.println(string.length));      // 7 → zero counting only when using indexes or positions within
                                         // a list; normal counting when determining total size / length
```

3.2.4.2 charAt()

- Used to query a string to find out what character is at a specific index.
- Signature: `char charAt(int index)`
- Examples:

```
String string = animals;
System.out.println(string.charAt(0));    // a
System.out.println(string.charAt(6));    // s
System.out.println(string.charAt(7));    // java.lang.StringIndexOutOfBoundsException
```

3.2.4.3 indexOf()

- Looks at characters in the string and finds the first index that matches the desired value.
- Can work with individual character or a whole `String` as input.
- Can start from a requested position.
- Signatures:
  - `int indexOf(int ch)`
  - `int indexOf(char ch, int fromIndex)`
  - `int indexOf(String str)`
  - `int indexOf(String str, index fromIndex)`
- Examples:

```
String string = animals;
System.out.println(string.indexOf('a');          // 0
System.out.println(string.indexOf("al"));        // 4
System.out.println(string.indexOf('a', 4));      // 4
System.out.println(string.indexOf("al" ', 5));   // -1 → doesn't throw exception if no match found,
                                                 // returns -1 instead
```

3.2.4.4 substring()

- Looks for characters in a string.
- Returns part of the string.
- First parameter is the index to start with for the returned string.
- Optional second parameter for end index you want to stop at (excluding that character), meaning `endIndex` parameter is allowed to be 1 past the end of the sequence.
- The `substring()` method is the trickiest `String` method on the exam.
- Signatures:
  - `String substring(int beginIndex)`
  - `String substring(int beginIndex, int endIndex)`
- Examples:

```
String string = "animals";
System.out.println(string.substring(3));                           // mals
System.out.println(string.substring(string.indexOf('m')));         // mals
System.out.println(string.substring(3, 4));                        // m
System.out.println(string.substring(3, 7));                        // mals

System.out.println(string.substring(3, 3));      // empty string
System.out.println(string.substring(3, 2));      // throws exception → indexes can't be backward
System.out.println(string.substring(3, 8));      // throws exception → length is 7, indexed from 0 to 6,
                                                 // endIndex is allowed to be 1 past end of sequence and
                                                 // no more: string.substring(3, 7) would be correct
```

- Recap: the method returns the string starting from the requested index. If an end index is requested, it stops right before that index (excluding the character at the end position). Otherwise, it goes to the end of the string.

3.2.4.5 toLowerCase() and toUpperCase()

- Convert strings to lower case and to upper case.
- Signatures:
  - `String toLowerCase()`
  - `String toUpperCase()`
- Examples:

```
String string = "animals";
System.out.println(string.toUpperCase());       // ANIMALS → converts any lowercase characters to upper-
                                                // case in the returned string
System.out.println("Abc123".toLowerCase());     // abc123 → converts any uppercase characters to lower-
                                                // case in the returned string
```

- Note: strings are immutable, so the original string stays the same.

3.2.4.6 equals() and equalsIgnoreCase()

- `equals()` checks whether two `String` objects contain exactly the same characters in the same order.
- `equalsIgnoreCase()` checks whether two `String` objects contain the same characters with the exception that it will convert characters' case if needed (it ignores differences in case).
- Signatures:
  - `boolean equals(Object obj)`
  - `boolean equalsIgnoreCase(Object obj)`
- Examples:

```
System.out.println("abc".equals("ABC"));                // false
System.out.println("ABC".equals("ABC"));                // true
System.out.println("abc".equalsIgnoreCase("ABC"));      // true
```

3.2.4.7 startsWith() and endsWith()

- These methods look at whether the provided value matches part of the `String`.
- The check is case-sensitive.

- Signatures:
  - `boolean startsWith(String prefix)`
  - `boolean endsWith(String postfix)`
- Examples:

```
System.out.println("abc".startsWith("a"));      // true
System.out.println("abc".startsWith("A"));      // false
System.out.println("abc".endsWith("c"));        // true
System.out.println("abc".endsWith("a"));        // false
```

3.2.4.8 *contains()*

- Looks for matches in the `String`.
- The match can be anywhere in the `String`.
- The check is case-sensitive.
- This is a convenience method so you don't have to write `str.indexOf(otherString) != -1`.
- Signature: `boolean contains(String str)`
- Examples:

```
System.out.println("abc".contains("b")); // true
System.out.println("abc".contains("B")); // false
```

3.2.4.9 *replace()*

- Does simple search and replace on the string.
- Takes `char` parameters as well as `CharSequence` parameters.
- A `CharSequence` is a general way of representing several classes, including `String` and `StringBuilder`.
- Signatures:
  - `String replace(char oldChar, char newChar)`
  - `String replace(CharSequence oldChar, CharSequence newChar)`
- Examples:

```
System.out.println("abcabc".replace('a', 'A')); // AbcAbc
System.out.println("abcabc".replace("a", "A")); // AbcAbc
```

3.2.4.10 *trim()*

- Removes whitespace from the beginning and end of a `String`.
- For the exam you need to know that whitespace consists of spaces along with the `\t` (tab) and `\n` (newline) characters as well as `\r` (carriage return).
- Signature: `public String trim()`
- Examples:

```
System.out.println("abc".trim());               // abc
System.out.println("\t   a b c\n".trim());      // a b c → leaves spaces in the middle of the string
```

3.2.5 Method Chaining

- It is common to call multiple methods on the same `String`.
- Examples:

```
String start = "AniMaL   ";
String trimmed = start.trim();                  // "AniMaL"→ returned value is put into new variable
String lowercase = trimmed.toLowerCase();       // "animal"→ returned value is put into new variable
String result = lowercase.replace('a', 'A');    // "AnimAl"→ returned value is put into new variable
System.out.println(result);

// method chaining → also creates four String objects; starts at the left, evaluates first method,
// then calls next method on returned value of first method and so on.
String result = "AniMaL   ".trim().toLowerCase().replace('a', 'A');
System.out.println(result);                     // "AnimAl"
```

3.3 Using the StringBuilder Class

- The `StringBuilder` class creates a `String` without storing all interim `String` values.
- `StringBuilder` is mutable.
- Example:

```
StringBuilder alpha = new StringBuilder();              // instantiation
for (char current = 'a'; current <= 'z'; current++)     // add character to the end of alpha
    alpha.append(current);
System.out.println(alpha);
```

3.3.1 Mutability and Chaining

- The exam will likely try to trick you with respect to `String` and `StringBuilder` being mutable.
- `StringBuilder` changes its own state and returns a reference to itself.
- Example:

```
StringBuilder sb = new StringBuilder("start");  // instantiation with value start
sb.append("+middle");                            // returns reference to sb → sb = "start+middle"
StringBuilder same = sb.append("+end");          // sb and same point to the exact same object →
                                                 // "start+middle+end"
```

3.3.2 Creating a StringBuilder

- Three ways of constructing a `StringBuilder`:

```
StringBuilder sb1 = new StringBuilder();       // contains empty sequence of characters assigned to sb1
StringBuilder sb2 = new StringBuilder("hi");   // contains specific value assigned to sb2
StringBuilder sb3 = new StringBuilder(10);     // reserves certain number of slots for characters
                                               // assigned to sb3
```

- Size vs. capacity:
  - The behind-the-scenes process of how objects are stored isn't on the exam.
  - Size is the number of characters currently in the sequence, capacity is the number of characters the sequence can currently hold.
  - `String` is immutable, therefore the size and capacity are the same.
  - Default capacity of `StringBuilder` is 16.
  - If the capacity of `StringBuilder` isn't large enough, Java automatically increases it.

3.3.3 Important StringBuilder Methods

3.3.3.1 charAt(), indexOf(), length(), and substring()

- These four methods work exactly the same as in the `String` class.
- `substring()` returns a `String` rather than a `StringBuilder`, so calling `substring()` on a `StringBuilder` does not change the value of the `StringBuilder`.

3.3.3.2 append()

- Adds the parameter to the `StringBuilder` and returns a reference to the current `StringBuilder`.
- One common signature: `StringBuilder append(String str)`
- There are more than 10 signatures for append, e.g. `append(1)`, `append(true)`, `append('c')`.
- In method chaining append can be directly called after the constructor: `StringBuilder sb = new StringBuilder.append("string1").append("string2")`.

3.3.3.3 insert()

- Adds characters to the `StringBuilder` at the requested index and returns a reference to the current `StringBuilder`.
- There are lots of signatures for different types, one common being: `StringBuilder insert(int offset, String str)`
- The requested parameter is inserted at the index position.

- Examples:

```
StringBuilder sb = new StringBuilder("animals");
sb.insert(7, "-");      // sb = animals-
sb.insert(0, "-");      // sb = -animals-
sb.insert(4, "-");      // sb = -ani-mals- → watch out on exam: when adding and removing characters, the
                        // indexes change
```

3.3.3.4 delete() and deleteCharAt()

- `delete()` removes characters from the sequence and returns a reference to the current `StringBuilder`.
- `deleteCharAt()` is convenient when you want to delete only one character.
- Signatures:
  - `StringBuilder delete(int start, int end) // start=including / end=excluding`
  - `StringBuilder deleteCharAt(int index)`
- Example:

```
StringBuilder sb = new StringBuilder("abcdef");
sb.delete(1, 3);        // sb = adef
sb.deleteCharAt(5);     // throws an exception (StringIndexOutOfBoundsException) → indexes changed in
                        // previous delete
```

3.3.3.5 reverse()

Reverses the characters in the sequence and returns a reference to the current `StringBuilder`.

3.3.3.6 toString()

- Converts a `StringBuilder` into a `String`.
- Signature: `String toString()`
- Often `StringBuilder` is used internally for performance purposes but the end result needs to be a `String`.

3.3.4 StringBuilder vs. StringBuffer

- When writing new code that concatenates a lot of `String` objects together, you should use `StringBuilder` (added in Java 5).
- In older code `StringBuffer` was often used. It does the same as `StringBuilder` but is slower because it is thread safe.

3.4 Understanding Equality

- For number and reference comparison == can be used.
- Examples:

```
StringBuilder one = new StringBuilder();
StringBuilder two = new StringBuilder();
StringBuilder three = one.append("a");
System.out.println(one == two);          // false → one and two refer to completely different
                                         // StringBuilders
System.out.println(one == three);        // true → StringBuilders return the current reference for
                                         // chaining; one and three point to the same object

String x = "Hello World";
String y = "Hello World";
System.out.println(x == y);     // true → string literals are pooled; since x and y are the same
                                // literals, only one is stored in the pool

String x = "Hello World";
String z = "Hello World".trim();
System.out.println(x == z);     // false → z is computed at runtime; it isn't the same at compile-time,
                                // a new String object is created

String x = new String("Hello World");
String y = "Hello World";
System.out.println(x == y);     // false → same result as the one above; for x a new String object is
                                // created
```

- You should never use == to compare String objects. It's better to use logical equality rather than object equality:

```
String x = "Hello World";
String z = " Hello World".trim();
System.out.println(x.equals(z)); // true → equals checks the values inside the String rather than the
                                 // String itself
```

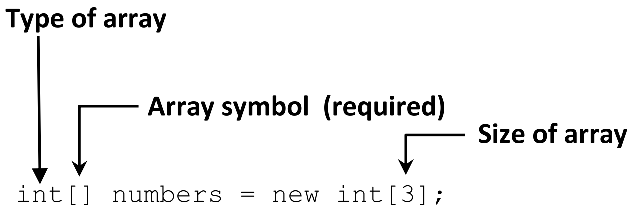
- If a class doesn't implement the `equals` method, Java determines whether the references point to the same object (same as using `==`).
- The `equals` method is not implemented in `StringBuilder`, therefore calling `equals()` on it will check for reference equality.

3.5 Understanding Java Arrays

- `String` and `StringBuilder` are implemented using an ***array*** of characters.
- An array is an area of memory on the heap with space for a designated number of elements.
- A `StringBuilder` is implemented as an array where the array object is replaced with a new bigger array object when it runs out of space to store all the characters.
- An array can be of any Java type.
- Instead of using a `String` we could use an array of `char` primitives: `char[] letters;`
  - `letters` is a reference variable, not a primitive
  - `char` is a primitive
  - `char` is what goes into the array and not the type of the array itself
  - the array is of type `char[]` → `[]` indicates an array
- An array is an ordered list.
- An array can contain duplicates.

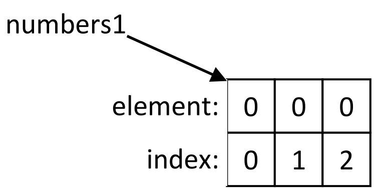
3.5.1 Creating an Array of Primitives

- Most common way to create an array: `int[] numbers1 = new int[3];`

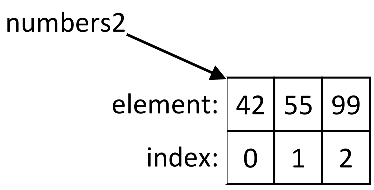


All elements are set to the default value for the type `int` (so 0).

- Array indexes start with 0:



- You can also specify all the elements at once: `int[] numbers2 = new int[] {42, 55, 99};`
- Here we also create an `int` array of size 3 where we specify the initial values:



- A shorter way to write the above is using an anonymous array. You don't specify the type and size, since Java already knows the type of initial values and also the size: `int[] numbers2 = {42, 55, 99};`

- You can type the [] before or after the name (adding a space is optional):

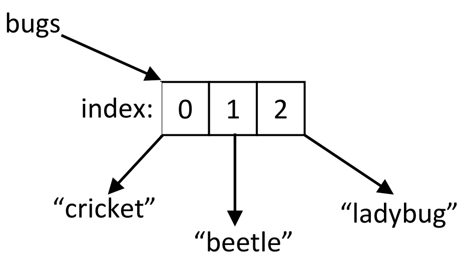
```
int[] numAnimals;      // most commonly used
int [] numAnimals2;
int numAnimals3[];
int numAnimals4 [];
```

- Example of multiple arrays in declarations:

```
int[] ids, types;      // two arrays of type int[]
int ids[], types;      // one variable of type int[] (ids) and one variable of type int (types)
```

3.5.2 Creating an Array with Reference Variables

- You can choose any Java type to be the type of the array. This includes classes you create yourself.
- The `equals()` method on arrays does not look at the elements of the array, it compares references.
- `array.toString()` prints something like `[Ljava.lang.String;@160bc7c0`
- Since Java 5 we can use `java.util.Arrays.toString(array)` to print the contents of the array nicely.
- An array (of type `String[]`) does not allocate space for the `String` objects, it allocates space for a reference to where the objects are actually stored:



- `String names[];` is a reference variable to `null` (it is not instantiated).
- `String names[] = new String[2];` is a reference variable to an array with two elements, each having a value of `null`.
- Watch out when casting a bigger type into a smaller type:

```
String[] strings = { "stringValue" };          // creates array of type String
Object[] objects = strings;                    // creates array of type Object and assigns strings to
                                               // it; no cast needed since Object is broader type than
                                               // String
String[] againStrings = (String[]) objects;    // creates array of type String and assigns objects array
                                               // to it; a cast to (String[]) is needed since we're
                                               // moving to more specific type and objects array happens
                                               // to contain an array of Strings, so this will work
againStrings[0] = new StringBuilder();         // this does not compile since a String[]can only contain
                                               // String objects
objects[0] = new StringBuilder();              // compiles: StringBuilder can go into an Object[], but
                                               // but we have String[] referred to Object[] variable;
                                               // this results in a RuntimeException
                                               //  (ArrayStoreException)
```

3.5.3 Using an Array

- Example:

```
String[] mammals = { "monkey", "chimp", "donkey" };    // declare and initialize array
System.out.println(mammals.length);                    // gives number of elements → 3 (not zero based)
System.out.println(mammals[0]);                        // access first element (zero based) → monkey
System.out.println(mammals[1]);                        // access second element → chimp
System.out.println(mammals[2]);                        // access first element → donkey

String[] birds = new String[6];
System.out.println(birds.length);                      // 6 (even if all elements are null → length does
                                                       // not consider what is in the array, only
                                                       // considers how many slots have been allocated
```

- **Note: the exam will test whether you are observant by trying to access elements that are not in the array (ArrayIndexOutOfBoundsException):**

```
int[] numbers = new int[10];
numbers[10] = 3;                           // indexes start from 0, so only 0 - 9 are valid
numbers[numbers.length] = 5;               // length is always one more than max. valid index;  this
                                           // results in the same (numbers[10] = 5) as previous example
for (int i = 0; i <= numbers.length; i++)  // loop should use < instead of <=
    numbers[i] = i + 5;
```

3.5.4 Sorting

- Almost any array can be sorted with `Arrays.sort()`.
- `Arrays` needs an import: `import java.util.*;` or `import java.util.Arrays;`
- If code snippet doesn't start with line number 1, assume the necessary import statements are there. Likewise if a snippet of a method is shown.
- Example:

```
int[] numbers = { 6, 9, 1 };
Arrays.sort(numbers);
for (int number : numbers) {
    System.out.print(number + " ");      // outputs 1 6 9
}

String[] strings = { "10", "9", "100" };
Arrays.sort(strings);
for (String string : strings) {
    System.out.print(string + " ");      // outputs 10 100 9 → Strings are sorted alphabetically (numbers
                                         // before letters, uppercase before lowercase)
}
```

3.5.5 Sorting

- We can search in arrays, but only if they are sorted.
- Binary search rules:

Scenario	Result
Target element found in sorted array	Index of match
Target element not found in sorted array	Negative value showing one smaller than the negative of index, where a match needs to be inserted to preserve sorted order
Unsorted array	A surprise – this result isn't predictable

- Example:

```
int[] numbers = { 2, 4, 6, 8 };                        // sorted array
System.out.println(Arrays.binarySearch(numbers, 2));   // 0 → 2 is found at index 0
System.out.println(Arrays.binarySearch(numbers, 4));   // 1 → 2 is found at index 1
System.out.println(Arrays.binarySearch(numbers, 1));   // -1 → 1 is not found, but to preserve order it
                                                       // should be inserted at index 0 negated - 1
                                                       // (0 - 1 is -1)
System.out.println(Arrays.binarySearch(numbers, 3));   // -2 → 3 not found, but to preserve order it
                                                       // should be inserted at index 1 negated - 1
                                                       // (-1 - 1 is -2)
System.out.println(Arrays.binarySearch(numbers, 9));   // -5 → 9 is not found, but to preserve order it
                                                       // should be inserted at index 4 negated - 1
                                                       // (-4 - 1 is -5)
```

- On the exam, as soon as you see the array isn't sorted, look for an answer choice about unpredictable output.

3.5.6 Varargs

- This means variable arguments.
- Example:

```
public static void main(String[] args);
public static void main(String args[]);
public static void main(String... args); // varargs definition → can be used as a normal array,
                                         // e.g. args.length and args[0] work on varargs
```

3.5.7 Multidimensional Arrays

Arrays are objects. Array components can be objects, so arrays can also hold other arrays.

3.5.7.1 Multidimensional Arrays

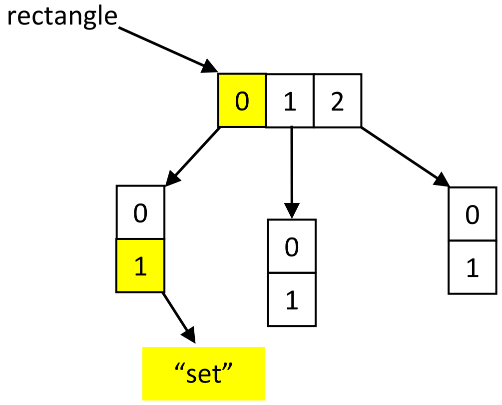
- Multiple array separators are all it takes to declare arrays with multiple dimensions.
- Examples:

```
int[][] vars1;               // 2D array
int vars2 [][];              // 2D array
int[] vars3[];               // 2D array → valid but confusing notation
int[] vars4 [], space[][];   // a 2D AND a 3D array
```

- You can specify the size of your multidimensional array in the declaration:

```
String[][] rectangle = new String[3][2]; // array with 3 elements each referring to array of 2 elements
                                          // addressable in range [0][0] through [2][1], e.g.:
rectangle[0][1] = "set";
```

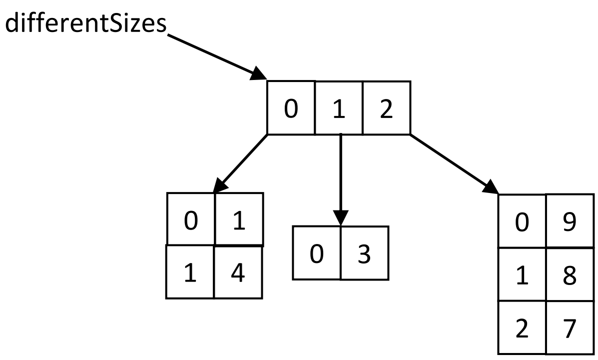
- Graphical representation of the above code:



- The number of elements within a next level of the array don't all have to be of the same size (asymmetric array), for example:

```
int[][] differentSizes = { { 1, 4 }, { 3 },  { 9, 8, 7 } };
```

- Graphical representation of the above multidimensional array:



- Another way to create an asymmetric array is to initialize just an array's first dimension, and define the size of each array component in a separate statement:

```
int [][] args = new int{4][];
args[0] = new int[5];
args[1] = new int[2];
```

3.5.7.2 *Using a Multidimensional Array*

- The most common operation on a multidimensional array is to loop through it using a basic `for` statement within a basic `for` statement where the first iterates over the outer array and the second iterates over the inner array, e.g.:

```
for (int i = 0; i < differentSizes.length; i++) {
    for (int j = 0; j < differentSizes[i].length; j++) {
       System.out.print(differentSizes[i][j] + " ");    // print element
    }
    System.out.println();                               // new line for next row
}

Outputs:
1 4
3
9 8 7
```

- The same can be accomplished with an enhanced `for` loop:

```
for (int[] inner : differentSizes) {
    for (int num : inner) {
       System.out.print(num + " ");      // print element
    }
    System.out.println();                // new line for next row
}
```

3.6 Understanding an ArrayList

- When creating an array, you need to know how many elements it will contain, so you're stuck with that choice.
- When using an `ArrayList`, you don't need to know the size beforehand. It can change size at runtime as needed.
- `ArrayList` requires an import: `import java.util.*;` or `import java.util.ArrayList;`

3.6.1 Creating an ArrayList

- Example:

```
ArrayList list1 = new ArrayList();       // ArrayList with default number of elements; no filled slots
ArrayList list2 = new ArrayList(10);     // ArrayList with specific number of slots, no filled slots
ArrayList list3 = new ArrayList(list2);  // ArrayList with copy of list2 (both size and contents)
```

- Since Java 5 we can use generics to specify the type of class that the `ArrayList` will contain:

```
ArrayList<String> list4 = new ArrayList<String>();      // Since Java 5
ArrayList<String> list5 = new ArrayList<>();            // Since Java 7 (diamond operator still needed)
```

- `ArrayList` implements an interface called `List`, so you can store an `ArrayList` in a `List` reference variable, but not vice versa:

```
List<String> list6 = new ArrayList<>();
ArrayList<String> list7 = new List<>();  // DOES NOT COMPILE
```

3.6.2 Using an ArrayList

- In method signatures, a "class" can be named `E`, which is used by convention in generics to mean "any class that this array can hold".
- If you didn't specify a type when creating the `ArrayList`, `E` means `Object`, otherwise it means the class you put between < and >.
- `ArrayList` implements `toString()`.

3.6.2.1 add()

- Inserts a new value in the `ArrayList`.
- Signatures:
  - `boolean add(E element)     // this always returns true`
  - `void add(int index, E element)`
- Examples:

```
ArrayList list = new ArrayList();// we don't specify a type, so we can put any Object (not primitves) in
                                 // the list!
list.add("hawk");                // [hawk]
list.add(Boolean.TRUE);          // [hawk, true]
System.out.println(list);        // [hawk, true]

List<String> safer = new ArrayList<>();  // now we specifiy that only String objects are allowed
safer.add("sparrow");
safer.add(Boolean.TRUE);                 // DOES NOT COMPILE

List<String> birds = new ArrayList<>();
birds.add("hawk");                       // [hawk]
birds.add(1, "robin");                   // [hawk, robin]
birds.add(0, "blue jay");                // [blue jay, hawk, robin]
birds.add(1, "cardinal");                // [blue jay, cardinal, hawk, robin]
System.out.println(birds);               // [blue jay, cardinal, hawk, robin]
```

- When a question has code that adds objects at indexed positions, draw it so that you won't lose track of which value is at which index.

3.6.2.2 remove()

- Removes the first matching value in the `ArrayList` or removes the element at a specified position.
- Signatures:
  - `boolean remove(Object object)    // boolean tells whether a match was removed`
  - `E remove(int index)              // E is return type of removed element`

- Examples:

```
ArrayList birds = new ArrayList();
birds.add("hawk");                              // [hawk]
birds.add("hawk");                              // [hawk, hawk]
System.out.println(birds.remove("cardinal"));   // prints false
System.out.println(birds.remove("hawk"));       // prints true
System.out.println(birds.remove(0));            // prints hawk
System.out.println(birds);                      // prints []
```

- When removing an element with an index that doesn't exist, an `IndexOutOfBoundsException` will be thrown.

3.6.2.3 set()

- Changes one of the elements of the `ArrayList` without changing the size.
- Signature:
  - `E set(int index, E newElement)`
- Examples:

```
ArrayList birds = new ArrayList();
birds.add("hawk");                              // [hawk]
System.out.println(birds.size());               // prints 1
System.out.println(birds.set(0, "robin"));      // [robin]
System.out.println(birds.size());               // prints 1
birds.set(1, "robin");                          // IndexOutOfBoundsException
```

3.6.2.4 isEmpty() and size()

- Look at how many of the slots are in use.
- Signatures:
  - `boolean isEmpty()`
  - `int size()`
- Examples:

```
System.out.println(birds.isEmpty());            // prints true
System.out.println(birds.size());               // prints 0
birds.add("hawk");                              // [hawk]
birds.add("hawk");                              // [hawk, hawk]
System.out.println(birds.isEmpty());            // prints false
System.out.println(birds.size());               // prints 2
```

3.6.2.5 clear()

- Discards all elements of the `ArrayList`, resulting in an empty `ArrayList` of size 0.
- Signature:
  - `void clear()`
- Examples:

```
List<String> birds = new ArrayList<>();
birds.add("hawk");      // [hawk] → isEmpty() is false / size() is 1
birds.add("hawk");      // [hawk] → isEmpty() is false / size() is 2
birds.clear();          // [] → isEmpty() is true / size() is 0
```

3.6.2.6 contains()

- Checks whether a certain value is in the `ArrayList`.
- Signature:
  - `boolean contains(Object object)`
- Examples:

```
List<String> birds = new ArrayList<>();
birds.add("hawk");                              // [hawk]
System.out.println(birds.contains("hawk"));     // prints true
System.out.println(birds.contains("robin"));    // prints false
```

- `contains()` calls `equals()` on each element of the `ArrayList` to see whether there are any matches. `String` implements `equals`, so we can do this for `ArrayList<String>`.

3.6.2.7 equals()

- `ArrayList` has a custom `equals` method implementation which can be used to compare two lists to see if they contain the same elements in the same order.
- Signature:
  - `boolean equals(Object object)`
- Examples:

```
List<String> one = new ArrayList<>();    // []
List<String> two = new ArrayList<>();    // []
System.out.println(one.equals(two));     // prints true
one.add("a");                            // [a]
System.out.println(one.equals(two));     // prints false
two.add("a");                            // [a]
System.out.println(one.equals(two));     // prints true
one.add("b");                            // [a, b]
two.add(0, "b");                         // [b, a]
System.out.println(one.equals(two));     // prints false → not in the same order
```

3.6.3 Wrapper Classes

- Each primitive type has a wrapper class, which is an object type that corresponds to the primitive:

Primitive type	Wrapper class	Example of constructing
`boolean`	`Boolean`	`new Boolean(true)`
`byte`	`Byte`	`new Byte((byte) 1)`
`short`	`Short`	`new Short((short) 1)`
`int`	`Integer`	`new Integer(1)`
`long`	`Long`	`new Long(1)`
`float`	`Float`	`new Float(1.0)`
`double`	`Double`	`new Double(1.0)`
`char`	`Character`	`new Character('c')`

- You don't need to know much about the constructors or `intValue()` type methods for the exam (autoboxing has removed the need for them).
- You need to be able to read the code though and not look for tricks in it.
- There are also methods for converting a `String` to a primitive or wrapper class (except for the `Character` class). You do need to know these methods:
  - Parse methods such as `parsInt()` return a primitive. The name of the returned primitive is in the method name, e.g.:

```
int primitive = Integer.parseInt("123");   // converts String "123" into int primitive 123
int bad1 = Integer.parseInt("a");          // throws NumberFormatException because String passed
                                           // for given type is not valid
```

  - The `valueOf()` method returns a wrapper class, e.g.:

```
Integer wrapper = Integer.valueOf("123");  // converts String "123" into Integer wrapper class
Integer bad2 = Integer.valueOf("123.45");  // throws NumberFormatException because String passed
                                           // for given type is not valid
```

- Converting from `String`:

Wrapper class	Converting String to primitive	Converting String to wrapper class
`Boolean`	`Boolean.parseBoolean("true");`	`Boolean.valueOf("TRUE");`
`Byte`	`Byte.parseByte("1");`	`Byte.valueOf("2");`
`Short`	`Short.parseShort("1");`	`Short.valueOf("2");`
`Integer`	`Integer.parseInt("1");`	`Integer.valueOf("2");`
`Long`	`Long.parseLong("1");`	`Long.valueOf("2");`
`Float`	`Float.parseFloat("1");`	`Float.valueOf("2.2");`
`Double`	`Double.parseDouble("1");`	`Double.valueOf("2.2");`
`Character`	None	None

3.6.4 Autoboxing

- Since Java 5 you can just type the primitive value and Java will convert it to the relevant wrapper class for you. This is called autoboxing.
- Examples:

```
List<Double> weights = new ArrayList<>();
weights.add(50.5);                          // [50.5]
weights.add(new Double(60));                // [50.5, 60.0]
weights.remove(50.5);                       // [60.0]
double first = weights.get(0);              // 60.0

weights.add(50);                            // fails: The method add(int, Double) in type List<Double> is
                                            // not applicable for arguments(int)

List<Integer> heights = new ArrayList<>();
heights.add(null);
int h = heights.get(0);                     // NullPointerException → any method call on null results in
                                            // NullPointerExeption; be careful when you see null in
                                            // relation to autoboxing.

List<Integer> numbers = new ArrayList<>();
numbers.add(1);                             // [1]
numbers.add(2);                             // [1, 2]
numbers.remove(1);                          // Be careful when autoboxing into Integer: remove method
                                            // takes an int parameter to remove the element based at the
                                            // index, thus it removes the element with value 2 and does
                                            // not autobox into Integer 1
System.out.println(numbers);                // prints 1
numbers.clear();                            // []
numbers.add(1);                             // [1]
numbers.add(2);                             // [1, 2]
numbers.remove(new Integer(1));             // To remove the item with the value 1, we can explicitly
                                            // create new Integer(1), or do numbers.remove(0)
System.out.println(number);                 // prints [2]
```

3.6.5 Converting Between array and List

- You should know how to convert between an `array` and an `ArrayList`.
- Converting an `ArrayList` into an `array`:

```
List<String> list = new ArrayList<>();
list.add("hawk");
list.add("robin");
Object[] objectArray = list.toArray();                // converts the list to an Object[]
System.out.println(objectArray.length);               // prints 2
String[] stringArray = list.toArray(new String[0]);   // specifies type of array we want; advantage of
                                                      // specifying size of 0 is that Java will create
                                                      // new array of proper size for the return value
System.out.println(stringArray.length);               // prints 2
```

- When converting an array into a `List`, the original array and the created array backed `List` are linked. A change made to one is also reflected in the other. It's a fixed-size list, known as a backed list because the array changes with it:

```
String[] array = { "hawk", "robin" };       // [hawk, robin]
List<String> list = Arrays.asList(array);   // returns fixed size list
System.out.println(list.size());            // prints 2
list.set(1, "test");                        // [hawk, test]
array[0] = "new";                           // [new, test]
for (String b : array) {
    System.out.print(b + "");               // prints new test
}
list.remove(1);                             // throws UnsupportedOperationException → we are not allowed
                                            // to change the size of the list because it is backed with
                                            // the array
```

- The `asList()` method can take varargs, which let you pass in an array or just type out the `String` values:

```
List<String> list = Arrays.asList("one", "two");
```

3.6.6 Sorting

- Sorting an `ArrayList` is very similar to sorting an array. You just use a different helper class:

```
List<Integer> numbers = new ArrayList<>();
numbers.add(99);
numbers.add(5);
numbers.add(81);
Collections.sort(numbers);
System.out.println(numbers);    // prints [5, 81, 99]
```

3.7 Working with Dates and Times

Note: the "old way" of working with dates (Date, Calendar) are not on the exam, but can still be used with Java 8.

3.7.1 Creating Dates and Times

- Note: creating dates and times with time zones is not covered on the exam.
- The following classes are covered on the exam, each of which has a static method `now()` which gives the current date and time as output depending on what date/time you run it with (based on where you live):
  - `LocalDate` is just a date without time and time zone, e.g.:

```
System.out.println(LocalDate.now());     // outputs 2015-01-20
```

  - `LocalTime` is a time without date and time zone, e.g.:

```
System.out.println(LocalTime.now());     // outputs 12:45:18:481
```

  - `LocalDateTime` is both date and time without time zone, e.g.:

```
System.out.println(LocalDateTime.now());   // outputs 2015-01-20T12:45:18:481 where Java uses T
                                           // to separate date and time when converting
                                           // LocalDateTime to a String
```

- Avoid using time zones unless strictly necessary. If you need them, `ZonedDateTime` can be used.
- On the exam date and time format will be in United States format, so just remember that month comes before the day. Java tends to use a 24-hour clock though, even if United States uses 12-hour clock with a.m./p.m.
- Another way to create a `LocalDate` is by using the static `of` method which has the following signatures:
  - `public static LocalDate of(int year, int month, int dayOfMonth)`
  - `public static LocalDate of(int year, Month month, int dayOfMonth)`

  Examples:

```
LocalDate date1 = new LocalDate.of(2015, 1, 20);             // uses int for month (non-zero based!)
LocalDate date2 = new LocalDate.of(2015, Month.JANUARY, 20);   // uses enum Month for month
```

- Another way to create a `LocalTime` is by using the static `of` method which has the following signatures:
  - `public static LocalTime of(int hour, int minute)`
  - `public static LocalTime of(int hour, int minute, int second)`
  - `public static LocalTime of(int hour, int minute, int second, int nanos)`

  Examples:

```
LocalTime time1 = new LocalTime.of(6, 15);             // hour and minutes
LocalTime time2 = new LocalTime.of(6, 15, 30);         // + seconds
LocalTime time3 = new LocalTime.of(6, 15, 30, 200);    // + nanoseconds
```

- `LocalDateTime` has the following signatures:
  - `public static LocalDateTime of(int year, int month, int dayOfMonth, int hour, int minute)`
  - `public static LocalDateTime of(int year, int month, int dayOfMonth, int hour, int minute, int second)`
  - `public static LocalDateTime of(int year, int month, int dayOfMonth, int hour, int minute, int second, int nanos)`
  - `public static LocalDateTime of(int year, Month month, int dayOfMonth, int hour, int minute)`
  - `public static LocalDateTime of(int year, Month month, int dayOfMonth, int hour, int minute, int second)`
  - `public static LocalDateTime of(int year, Month month, int dayOfMonth, int hour, int minute, int second, int nanos)`
  - `public static LocalDateTime of(LocalDate date, LocalTime time)`

  Examples:

```
LocalDateTime dateTime1 = new LocalDateTime.of(2015, Month. JANUARY, 20, 6, 15, 30);
LocalDateTime dateTime2 = new LocalDateTime.of(date1, time1);  // we can create LocalDate and LocalTime
                                                               // separately and use them to construct a
                                                               // LocalDateTime
```

- The date and time classes have a private constructor to force you to use the static methods. Therefore:
  `LocalDate d = new LocalDate();`
  will result in a compilation error.
- `LocalDate.of(2015, Month.JANUARY, 32);` will result in a `DateTimeException`.

3.7.2 Manipulating Dates and Times

- The date and time classes are immutable, therefore remember to assign the results of these methods to a reference variable so they are not lost, e.g.:

```
LocalDate date = LocalDate.of(2014, Month. JANUARY, 20);
date = date.plusDays(2);         // adds two days to the original date and reassigns the result to date
                                 // other methods could be: plusWeeks(2), plusMonths(1), plusYears(5) or
                                 // minusDays(2), minusWeeks(2), minusMonths(2), minusYears(5)
```

- Java automatically takes care of leap years when adding days, weeks, months or years.
- When using `LocalDateTime` we can also add hours (`plusHours(1)`), minutes (`plusMinutes(2)`), seconds (`plusSeconds(10)`), and nanoseconds (`plusNanos(100)`) as well as go backward in time (`minusHours(1)`, `minusMinutes(2)`, `minusSeconds(10)`, `minusNanos(100)`), e.g.:

```
LocalDate date = LocalDate.of(2020, Month. JANUARY, 20);
LocalTime time = LocalTime.of(5, 15);
LocalDateTime dateTime = LocalDateTime.of(date, time);
dateTime = dateTime.minusHours(1);      // subtracts one hours from original dateTime and reassigns the
                                        // result to dateTime
System.out.println(dateTime);           // displays 2020-01-20T04:15
dateTime = dateTime.minusSeconds(10);   // subtracts 10 seconds from dateTime and reassigns the result to
                                        // dateTime
System.out.pringln(dateTime);           // displays 2020-01-20T04:14:50 → note that dateTime was created
                                        // without seconds, so initially they are not displayed, but once
                                        // used it Java will show seconds and nanoseconds
```

- Date and time methods can be chained, e.g.:

```
LocalDate date = LocalDate.of(2014, Month. JANUARY, 20);
LocalTime time = LocalTime.of(5, 15);
LocalDateTime dateTime = LocalDateTime.of(date, time).minusDays(1).minusHours(10).minusSeconds(30);
```

- **Watch out for the following on the exam:**

```
LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
date.plusDays(10);
System.out.println(date);        // will display 2020-01-20 and not 2020-01-30, because the result of the
                                 // of operation was not reassigned to date
```

- **Also watch out for the following on the exam:**

```
LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
date = date.plusMinutes(1);      // DOES NOT COMPILE because LocalDate cannot contain time, so you cannot
                                 // do time manipulation on it like adding minutes; this can be tricky in
                                 // a chained sequence of additions/subtraction operations
```

- Table of allowed methods for `LocalDate`, `LocalTime` and `LocalDateTime`:

Wrapper class	Can call on LocalDate?	Can call on LocalTime?	Can call on LocalDateTime?
`plusYears/minusYears`	Yes	No	Yes
`plusMonths/minusMonths`	Yes	No	Yes
`plusWeeks/minusWeeks`	Yes	No	Yes
`plusDays/minusDays`	Yes	No	Yes
`plusHours/minusHours`	No	Yes	Yes
`plusMinutes/minusMinutes`	No	Yes	Yes
`plusSeconds/minusSeconds`	No	Yes	Yes
`plusNanos/minusNanos`	No	Yes	Yes

3.7.3 Working with Periods

- `LocalDate` and `LocalDateTime` have a method to convert them into `long` equivalents in relation to 1970 (which UNIX started using for date standards).
- `LocalDate` has `toEpochDay()`, which is the number of days since January 1, 1970.
- `LocalDateTime` has `toEpochSecond()`, which is the number of seconds since January 1, 1970.
- The time since January 1, 1970 is in GMT (Greenwhich Mean Time).

- `Period` has the following signatures:
    - `public static Period ofYears(int years)`
    - `public static Period ofMonths(int months)`
    - `public static Period ofWeeks(int weeks)`
    - `public static Period ofDays(int days)`
    - `public static Period of(int years, int months, int days)`

  Examples:

```
Period annually = Period.ofYears(1);              // every 1 year
Period quarterly = Period.ofMonths(3);            // every 3 months
Period everyThreeWeeks = Period.ofWeeks(3);       // every 3 weeks
Period everyOtherDay = Period.ofDays(2);          // every 2 days
Period everyYearAndAWeek = Period.of(1, 0, 7);    // every year and 7 days
```

- You cannot chain methods when creating a `Period`. `Period.of(XXX)` are static methods, so only the last method is used:

```
Period wrong = Period.ofYears(1).ofWeeks(1);      // results in every week and not in everyYearAndAWeek;
                                                  // it's the same as writing:
                                                  //    Period wrong = Period.ofYears(1);
                                                  //    wrong = Period.ofWeeks(1);
                                                  // you will get a compiler warning
```

- A `Period` is a day or more of time.
- `Duration` is intended for smaller units of time where you can specify number of days, hours, minutes, seconds or nanoseconds. **`Duration` isn't on the exam!**
- What objects can `Period` be used with?

```
LocalDate date = LocalDate.of(2015, 1, 20);
LocalTime time = LocalTime.of(6, 15);
LocalDateTime dateTime = LocalDateTime.of(date, time);
Period period = Period.ofMonths(1);
System.out.println(date.plus(period);        // 2015-02-20 → as expected: adds 1 month to date
System.out.println(dateTime.plus(period);    // 2015-02-20T06:15 → as expected: adds 1 month to dateTime
System.out.println(time.plus(period);        // UnsupportedTemporalTypeException → adding a month to a
                                             // time object is not allowed
```

3.7.4 Formatting Dates and Times

- `DateTimeFormatter` from the `java.time.format` package can be used to format any type of date and/or time object.
- Examples of `DateTimeFormatter` using ISO standard for displaying date/time:

```
LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
LocalTime time = LocalTime.of(11, 12, 34);
LocalDateTime dateTime = LocalDateTime.of(date, time);
System.out.println(date.format(DateTimeFormatter.ISO_LOCAL_DATE));          // 2020-01-20
System.out.println(time.format(DateTimeFormatter.ISO_LOCAL_TIME));          // 11:12:34
System.out.println(dateTime.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME)); // 2020-01-20T11:12:34
```

- Examples of `DateTimeFormatter` using predefined formats (**two of which can be expected on the exam are SHORT and MEDIUM**):

```
DateTimeFormatter shortDateTime = DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);
DateTimeFormatter mediumDateTime = DateTimeFormatter.ofLocalizedDate(FormatStyle.MEDIUM);
System.out.println(shortDateTime.format(dateTime));    // 1/20/20
System.out.println(mediumDateTime.format(dateTime));   // Jan 20, 2020 11:12:34 AM
System.out.println(shortDateTime.format(date));        // 1/20/20
System.out.println(shortDateTime.format(time));        // UnsupportedTemporalTypeException → time
                                                       // cannot be formatted as a date
```

- The `format()` method is declared both on the formatter objects and the date/time objects, allowing you to reference objects in either order, so the following will result in exactly the same as the examples above:

```
DateTimeFormatter shortDateTime = DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);
DateTimeFormatter mediumDateTime = DateTimeFormatter.ofLocalizedDate(FormatStyle.MEDIUM);
System.out.println(dateTime.format(shortDateTime));    // 1/20/20
System.out.println(dateTime.format(mediumDateTime));   // Jan 20, 2020 11:12:34 AM
System.out.println(date.format(shortDateTime));        // 1/20/20
System.out.println(time.format(shortDateTime));        // UnsupportedTemporalTypeException → time
                                                       // cannot be formatted as a date
```

- `ofLocalized` method:

DateTimeFormatter f = DateTimeFormatter.___ (FormatStyle.SHORT);	Calling f.format (localDate)	Calling f.format (localDateTime)	Calling f.format (localTime)
`ofLocalizedDate`	Legal – shows whole object	Legal – shows just date part	Throws runtime exception
`ofLocalizedDateTime`	Throws runtime exception	Legal – shows whole object	Throws runtime exception
`ofLocalizedTime`	Throws runtime exception	Legal – shows just time part	Legal – shows whole object

- You can create your own format (**for the exam you're not expected to memorize what different numbers of each symbol mean**):

```
DateTimeFormatter f = DateTimeFormatter.ofPattern("MMMM dd, yyyy, hh:mm");   // M for month, d for day,
                                                                              // y for year, h for hour,
                                                                              // m for minute
System.out.println(dateTime.format(f));          // January 20, 2020, 11:12
```

- When having a formatter containing only time, we can only use it on objects containing time and when having a formatter containing only date, we can only use it on objects containing a date:

```
DateTimeFormatter ft = DateTimeFormatter.ofPattern("hh:mm");
System.out.println(ft.format(dateTime));     // 11:12
System.out.println(ft.format(time));         // 11:12
System.out.println(ft.format(date));         // UnsupportedTemporalTypeException → formatter which only
                                             // contains time cannot be used on a date field

DateTimeFormatter fd = DateTimeFormatter.ofPattern("MMMM dd, yyyy");
System.out.println(fd.format(dateTime));     // January 20, 2020
System.out.println(fd.format(date));         // January 20, 2020
System.out.println(fd.format(time));         // UnsupportedTemporalTypeException → formatter which only
                                             // contains date cannot be used on a time field
```

3.7.5 Parsing Dates and Times

- With the `format()` method we can pass date and/or time to a `String`. The `parse()` method, which also takes a formatter, allows us to convert a `String` to a date and/or time object. If no formatter is specified, the default for the type is used:

```
DateTimeFormatter f = DateTimeFormatter.ofPattern("MM dd yyyy");
LocalDate date = LocalDate.parse("01 02 2015", f);
LocalTime time = LocalTime.parse("11:22");
System.out.println(date);                        // 2015-01-02
System.out.println(time);                        // 11:22
```

- If anything goes wrong, e.g. providing an invalid `String` not matching format, Java throws runtime exception.

3.8 Exam Essentials

- ***Be able to determine the output of code using String:*** `Strings` are immutable, indexes are zero based, `substring()` gets string up until right before the index of the second parameter.
- ***Be able to determine the output of code using StringBuilder:*** `StringBuilder` is mutable, `substring()` does not change value of `StringBuilder` but `append()`, `delete()` and `insert()` do, the methods return a reference to the current instance.
- ***Understand the difference between == and equals:*** `==` checks object equality, `equals()` depends on the implementation of the object it is being called on.
- ***Be able to determine the output of code using arrays:*** know how to declare and instantiate one- and multidimensional arrays, how to access each element, when index is out of bounds and recognize correct and incorrect output when searching and sorting.
- ***Be able to determine the output of code using ArrayList:*** `ArrayList` can increase in size, know different ways of declaring and instantiating, identify correct output from its methods, including impact of autoboxing.
- ***Recognizing invalid uses of dates and times:*** `LocalDate` has no time fields, `LocalTime` has no date fields, watch out for operations on wrong time, watch out when adding/subtracting time and ignoring the result.

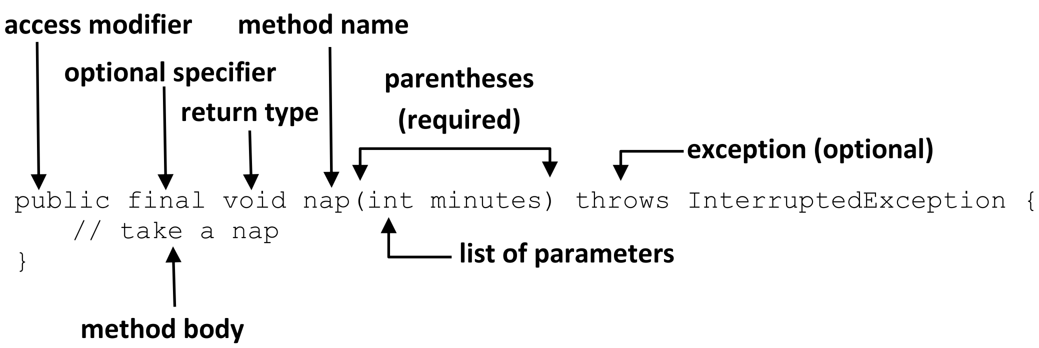
4 Methods and Encapsulation

4.1 OCA Exam objectives

1. Working with Methods and Encapsulation:
   a. Create methods with arguments and return values, including overloaded methods.
   b. Apply static keyword to methods and fields.
   c. Create and overload constructors; include impact on default constructors.
   d. Apply access modifiers.
   e. Apply encapsulation principles to a class.
   f. Determine effect upon object references and primitive values when they are passed into methods that change the values.
2. Working with Selected classes from the Java API:
   a. Write simple Lambda expression that consumes a Lambda Predicate expression.

4.2 Designing Methods

- The following figure depicts a method declaration, which specifies all the information needed to call a method:



Element	Value in nap() example	Required?
Access modifier	`public`	No
Optional specifier	`final`	No
Return type	`void`	Yes
Method name	`nap`	Yes
Parameter list	`(int minutes)`	Yes
Optional exception list	`throws InterruptedException`	No
Method body	`{ // take a nap }`	Yes, but can be empty braces

- To call the above method, just type its name, followed by a single `int` value in parentheses, e.g. `nap(10);`

4.2.1 Access Modifiers

- Four access modifiers:

Access Modifier	Description
`public`	can be called from any class
`private`	can only be called from within the same class
`protected`	can only be called from classes in same package or subclasses
Default (Package Private) access	can only be called from classes in the same package (there is no keyword for this, simply omit access modifier); there is a `default` keyword used in `switch` statement and interfaces, but not used for access control!

- The exam creators like to trick you by putting method elements in the wrong order or using incorrect values.

```
public void walk() {}     // valid method declaration with public access modifier
default void walk2() {}   // DOES NOT COMPILE: default is not a valid access modifier
void public walk3() {}    // DOES NOT COMPILE: wrong order of access modifier and return type
void walk4() {}           // valid method declaration with default (package private) access
```

4.2.2 Optional Specifiers

- You can have multiple optional specifiers in the same method (not all combinations are legal) and they can be specified in any order (most of them are not on the exam):

Optional Specifier	Description
`static`	used for class methods
`abstract`	used when not providing a method body
`final`	used when method is not allowed to be overridden by a subclass
`synchronized`	not on OCA (but on OCP)
`native`	used when interacting with code written in another language as C++ (not on OCA or OCP)
`strictfp`	used for making floating-point calculations portable (not on OCA or OCP)

- Examples:

```
public void walk1() {}             // valid without optional specifier
public final void walk2() {}       // valid with final as optional specifier
public static final void walk3() {} // valid with static and final as optional specifiers (preferred
                                   // order)
public final static void walk4() {} // valid with final and static as optional specifiers
public modifier void walk5() {}    // DOES NOT COMPILE: modifier is not a valid optional specifier
public void final walk6() {}       // DOES NOT COMPILE: wrong order optional specifier and return type
final public void walk7() {}       // valid with optional specifier final; optional specifiers are
                                   // allowed to appear in front of access modifiers
```

4.2.3 Return Type

- Return type can be actual Java type, like `String` or `int`.
- When no return type is necessary, the keyword `void` is used, meaning without contents.
- A method must have a return type, it cannot be omitted.
- Methods with a return type other than `void` are required to have a `return` statement inside the method body and must include the primitive or object to be returned.
- In case of `void` return type, the method body can have either a `return` statement with no value returned or no return statement at all.
- Examples:

```
public void walk1() { }             // valid: void return type, return statement is optional
public void walk2() { return; }     // valid: void return type, return statement without value
public String walk3() { return “”; } // valid: String return type, String value is returned
public String walk4() { }           // DOES NOT COMPILE: return type String, but nothing is returned
public walk5() { }                  // DOES NOT COMPILE: no return type specified
String walk6(int a) {               // DOES NOT COMPILE: return type String, but a String is only
    if (a == 4)                     // returned in a certain condition if a == 4; nothing is returned
       return “”;                   // if the condition a == 4 is not met
}
```

- When returning a value, it needs to be assignable to the return type:

```
int integer() { return 9; }     // return type int, int is returned
int longMethod() { return 9L; } // DOES NOT COMPILE: return type int, a long is returned
```

4.2.4 Method Names

- Method names follow the same rules as variable names:
  - may only contain letters, numbers, $, or _
  - first character is not allowed to start with a number
  - reserved keywords not allowed
- Convention: methods begin with lower case letter.
- Examples:

```
public void walk1() {}    // valid method declaration, traditional name
public void 2walk() {}    // DOES NOT COMPILE: cannot start with a number
public walk3 void() {}    // DOES NOT COMPILE: wrong order, method name cannot be before return type
public void walk_$() {}   // valid method declaration
public void() {}          // DOES NOT COMPILE: method name is missing
```

4.2.5 Parameter List

- Parameter list is required but can be empty (empty pair of parentheses after method name).
- Multiple parameters are separated by a comma.
- Examples:

```
public void walk1() {}              // valid with no parameters
public void walk2 {}                // DOES NOT COMPILE: parentheses missing
public void walk3(int a) {}         // valid with one parameter
public void walk4(int a; int b) {}  // DOES NOT COMPILE: multiple parameters should be separated by comma
public void walk5(int a, int b) {}  // valid with two parameters separated by comma
```

4.2.6 Optional Exception List

- You can list as many types of exceptions as you want separated by commas.
- The calling method can throw the same exceptions or handle them.
- Examples:

```
public void zeroExceptions() {}
public void oneException() throws IllegalArgumentException {}
public void twoExceptions() throws IllegalArgumentException, InteruptedExcetpion {}
```

4.2.7 Method Body

- A method declaration has a method body (except for abstract methods and interfaces).
- A method body is simply a code block with zero or more Java statements.
- Examples:

```
public void walk1() {}                      // valid with empty method body
public void walk2();                        // DOES NOT COMPILE: missing braces
public void walk3(int a) { int name = 5; }  // valid with one statement in the body
```

4.3 Working with Varargs

- A vararg parameter must be the last element in a method's parameter list.
- You can only have one vararg parameter per method.
- Examples:

```
public void walk1(int... nums) {}              // valid, with one vararg
public void walk2(int start, int... nums) {}   // valid, with two parameters, the last being a vararg
public void walk3(int... nums, int start) {}   // DOES NOT COMPILE: vararg should be the last parameter
public void walk4(int... start, int... nums) {} // DOES NOT COMPILE: cannot have more than one
                                               // vararg parameter
```

- Calling a method with vararg can be done either by passing in an array or by listing the elements and have Java create the array for you.
- When omitting the vararg, Java creates an array of length zero.
- Examples:

```
public static void walk(int start, int... nums) {
    System.out.println(nums.length);
}

public static void main(String[] args) {
    walk(1);                        // 0 → passes 1 for start, Java creates zero length array
    walk(1, 2);                     // 1 → passes 1 for start and 2 for varargs (array of length 1)
    walk(1, 2, 3);                  // 2 → passes 1 for start and varargs with two int values (array
                                    //      length of 2)
    walk(1, new int[] {4, 5]});     // 2 → passes 1 for start and an array with two ints (array length
                                    //      of 2)
    walk(1, null);                  // it is possible to pass null; this throws a NullPointerException
}
```

- Accessing vararg parameter is like accessing an array, using array indexing.

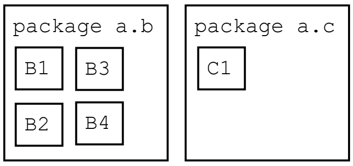
4.4 Applying Access Modifiers

- Access modifiers from most restrictive to least restrictive:

Access Modifier	Description
`private`	only accessible within the same class
`default (package private)`	private and other classes in same package
`protected`	default access and child classes
`public`	protected and classes in other packages

4.4.1 Private Access

- Accessing private members (instance variables and methods) of other classes is not allowed.
- Classes to demonstrate `private` and default access:



- Legal example using private access only (everything in one class):

```
package a.b;
public class B1 {
    private String letter = “b”;   // private access
    private void print() {
       System.out.println(letter); // private access is ok
    }
    private void printLetter() {
       print();                    // private access is ok; B1 makes call to private method print in B1,
    }                              // which in turn uses private instance variable letter to print
}
```

- But:

```
package a.b;
public class B2 {                     // another class B2 in the same a.b package as class B1
    public void printLetter() {
       B1 b1 = new B1();              // we can create a new instance of B1 since the class is public
                                      // and has a default no-argument constructor
       b1.print();                    // DOES NOT COMPILE: class B2 cannot access private method of an
                                      //                   instance of class B1
       System.out.println(b1.letter); // DOES NOT COMPILE: class B2 cannot directly access a private
    }                                 //                   instance variable of class B1
}
```

4.4.2 Default (Package Private) Access

- This allows classes in the same package to access her members:

```
package a.b;
public class B3 {
    String letter = “b”;           // default access
    void print() {
       System.out.println(letter); // default access is ok
    }
    private void printLetter() {
       print();                    // default access is ok; B3 can call print and refer to letter;
    }                              // members in the same class are also in the same package
}
```

- And B3 lets other classes in the same package access members:

```
package a.b;
public class B4 {                     // another class B4 in the same a.b package as class B3
    public void printLetter() {
       B3 b3 = new B3();              // we can create a new instance of B3 since the class is public
                                      // and has a default no-argument constructor
       b3.print();                    // default access, so class B4 can access default method of an
                                      // instance of class B3 in the same package
       System.out.println(b3.letter); // default access, so class B4 can directly access a default
    }                                 // instance variable of class B3 in the same package
}
```

- But:

```
package a.c;
import a.b.B3;                           // import class B3 from another package
public class C1 {                        // another class C1 in a different package a.b as class B3
    public void printLetter() {
       B3 b3 = new B3();                 // we can create a new instance of B3 since the class is public
                                         // and has a default no-argument constructor
       b3.print();                       // DOES NOT COMPILE: class C1 from another package cannot access
                                         // default method in an instance of class B3 in another package
       System.out.println(b3.letter);    // DOES NOT COMPILE: class C1 from another package cannot directly
                                         // access a default instance variable of class B3 in another
    }                                    // package
}
```

4.4.3 Protected Access

- Protected access allows everything that default access allows and more. It adds the ability to access members of a parent class.
- Classes to demonstrate `protected` and `public` access:

package a.b	package a.c	package a.d	package a.e	package a.f
B1 B2	C1 C2 C3	D1	E1	F1 F2

- Valid and invalid examples of protected access:

```
package a.b;
public class B1 {
  protected String letter = "b";     // protected access
  protected void printLetter() {
     System.out.println(letter);     // protected access
  }
}

package a.c;
import a.b.B1;                       // import class B1 from another package than C1
public class C1 extends B1 {         // extends means create a subclass which has access to any protected
                                     // or public member of the parent class B1
  public void doPrint() {
     printLetter();                  // calling the protected method from the B1 superclass is valid,
                                     // even when it is in a different package
     System.out.println(letter);     // calling protected instance variable from B1 superclass is valid,
  }                                  // even when it is in a different package
}

package a.b;
public class B2 {                    // another class B2 in the same package as class B1
  B1 b1 = new B1();                  // creates a new instance of class B1
  b1.printLetter();                  // calling protected method from B1 allowed, even though class B1
                                     // does not extend class B2, but since both classes are in the same
                                     // package B2 can access members of the b1 instance variable
  System.out.println(b1.letter);     // calling protected instance variable from B1 allowed (same
}                                    // explanation as statement above)

package a.d;
import a.b.B1;                       // import class B1 from another package than D1
public class D1 {
  public void doPrint() {
     B1 b1 = new B1();               // creates a new instance of class B1
     b1.printLetter();               // DOES NOT COMPILE: since B1 is not in the same package as D1 and D1
                                     //                   doesn't inherit from B1, D1 cannot access the
                                     //                   protected members of B1
     System.out.println(b1.letter);  // DOES NOT COMPILE: same explanation as statement above
  }
}

package a.e;
import a.b.B1;                       // import class B1 from another package than E1
public class E1 extends B1 {         // E1 is a subclass of B1
  public void doPrint() {
     printLetter();                  // member used without referring to variable: calling protected
                                     // method from B1 superclass is valid, even when it is in different
                                     // package
     System.out.println(letter);     // member used without referring to variable: calling protected
                                     // instance variable from B1 superclass is valid, even when it is in
  }                                  // a different package
```

```
  // here we are in the E1 class and we refer to an E1 instance variable
  // therefore we can access the protected members of the superclass
  public void doPrintE1() {
     E1 e1 = new E1();              // creates a new instance of class E1
     e1.doPrint();                  // member used through variable: E1 extends B1, so e1 instance
                                    // variable IS-A B1, so calling protected method through e1 is valid
     System.out.println(e1.letter); // member used through variable: E1 extends B1, so e1 instance
                                    // variable IS-A B1, so calling protected instance variable through
  }                                 // e1 is valid

  public void doPrintOtherB1() {
     B1 other = new B1();           // creates a new instance of class B1
     other.printLetter();           // DOES NOT COMPILE: a member is used through a variable: B1 is in a
                                    //                   different package and doesn't inherit from B1,
                                    //                   therefore it is not allowed to use protected
                                    //                   members
     System.out.println(b1.letter); // DOES NOT COMPILE: same explanation as statement above
  }

package a.c;
import a.b.B1;                      // import class B1 from another package than C2
public class C2 extends B1 {        // extends means create a subclass which has access to any protected
                                    // or public member of the parent class B1

  // this method is like the method doPrintE1() in the E1 class
  // here we are in the C2 class and we refer to a C2 instance variable
  // therefore we can access the protected members of the superclass
  public void doPrintC2() {
     C2 c2 = new C2();              // creates a new instance of class C2
     c2.printLetter();              // member used through variable: C2 extends B1, so c2 instance
                                    // variable IS-A B1, so calling protected method through c2 is valid
     System.out.println(c2.letter); // member used through variable: C2 extends B1, so c2 instance
                                    // variable IS-A B1, so calling protected instance variable through
  }                                 // c2 is valid

  // this is a problem: we create a C2 object, but we store it in a B1 reference
  // we are not allowed to refer to members of the B1 class since we are not in the same package
  // and B1 is not a subclass of C2
  public void doPrintOtherC2() {
     B1 b1 = new C2();              // creates a new instance of C2, but assigns it to its superclass
                                    // instance variable of type B1
     b1.printLetter();              // DOES NOT COMPILE: cannot refer to member of B1 since B1 is not
                                    //                   in the same package as C2 and B1 is not a
                                    //                   subclass of C2
     System.out.println(b1.letter); // DOES NOT COMPILE: same explanation as statement above
  }
}

package a.f;
import a.c.C2;
public class F1 {
  public void doPrint() {
     C2 c2 = new C2();              // creates new instance of C2 which is in another package than F1
     c2.printLetter();              // DOES NOT COMPILE: we are not in the C2 class; printLetter() is
                                    //                   declared in B1, but C2 is not in the same
                                    //                   package as B1, nor does it extend B1, thus only
                                    //                   C2 is allowed to refer to printLetter() and not
  }                                 //                   callers of C2
}
```

- Protected rules apply under two scenarios (as seen in the examples above):
  - Member used without referencing variable (taking advantage of inheritance; `protected` access allowed).
  - Member used through variable. Here the rules for the reference type of the variable are what matter. If it is a subclass, protected access is allowed. This works for references to the same class or a subclass.

4.4.4 Public Access

- When using `public`, anyone can access the member from anywhere.
- Example:

```
package a.f;
public class F2 {
    public String letter = "f";         // public access
    public void print() {
       System.out.println("print");     // public access
    }
}
```

```
package a.c;
import a.f.F2;                      // import class F2 from another package than C3
public class C3 {
   public void print() {
      F2 f2 = new F2();             // creates a new instance of class F2
      f2.print();                   // print method in F2 has public access so this is allowed
      System.out.println("letter: " // instance variable letter in F2 has public access so this is
         + f2.letter);              // allowed
   }
}
```

- Access Modifiers reviewed:

Can access	If that member is private?	If that member has default (package private) access?	If that member is protected?	If that member is public?
Member in same class	Yes	Yes	Yes	Yes
Member in another class in same package	No	Yes	Yes	Yes
Member in a superclass in different package	No	No	Yes	Yes
Method/field in a non-superclass in a different package	No	No	No	Yes

4.4.5 Designing Static Methods and Fields

- Static methods don't require an instance of the class.
- They are shared among all users of the class.
- Each class has a copy of the instance variables but there is only one copy of the code for the instance methods, which can be called by each instance of the class as many times as it would like.
- Each call of instance method (or any method) gets space on the stack for method parameters and local variables.
- Same happens for static methods: there's only one copy of the code; parameters and local variables go on the stack.
- Only data (instance variables) gets its "own copy". There is no need to duplicate copies of the code itself.
- Static methods have two main purposes:
  - For utility or helper methods that don't require object state, so no need for instance variables; static methods eliminate the need for the caller to instantiate the object just to call the method.
  - For state that is shared by all instances of a class, like a counter. Methods merely using that state should be static as well.

4.4.6 Calling a Static Variable or Method

- A static method can be called through its classname, e.g. if class `A` has a static method `b`, it can be called like `A.b();` and if class `A` has a static variable `c`, it can be called like `A.c;`
- You can also use an instance of the object to call a static method, e.g. if class `A` has a static variable `c`, we can do:
  `A a = new A(); System.out.println(a.c); a = null; System.out.println(a.c);`
  Java doesn't care if `a` is `null` in this case, because it looks for a static.
  Remember to look at the reference type for a variable when you see a static method or variable. Exam creators try to trick you in believing the above example causes a `NullPointerException`.
- Example:

```
A.c = 4;
A a1 = new A();
A a2 = new A();
a.c = 6;
a.c = 5;
System.out.println(A.c);  // prints 5; c is static, there is only one copy which is set to 4, then 6,
                          // then 5
```

4.4.7 Static vs. Instance

- A static member (field or method) cannot call an instance member (static doesn't require any instances). The exam tries to trick you here:

```
public class Static {
    private String name = "Static class";
    public static void first() {}
    public static void second() {}
    public void third {
       System.out.println(name);
    }
    public static void main(String args[]) {
       first();         // static method can call another static method
       second();        // static method can call another static method
       third();         // DOES NOT COMPILE: static method cannot call an instance (nonstatic) method
    }
}
```

- A static method or instance method can call a static method since static methods don't require an object to use.
- Only an instance method can call another instance method on the same class without using a reference variable, because instance methods do require an object. Similar logic applies for the instance and static variables.
- Static vs. instance calls:

Type	Calling	Legal?	How?
Static method	Another static method or variable	Yes	Using the classname
Static method	An instance method or variable	No	Not without instantiating the object
Instance method	A static method or variable	Yes	Using the classname or a reference variable
Instance method	Another instance method or variable	Yes	Using a reference variable

4.4.8 Static Variables

- Static variables can change as the program runs like counters for counting the number of instances for example.
- Other static variables (constants) never change during the run of the program; these use the `final` modifier.
- These `static final` constants use a different naming convention than other variables: all uppercase letters with underscores between "words":

```
public class Initializers {
    private static final int NUM_BUCKETS = 45;
    public static void main(String[] args) {
       NUM_BUCKETS = 5;    // DOES NOT COMPILE: static final (constants) cannot be changed
}
```

- But:

```
public class Initializers {
    private static final ArrayList<String> values = new ArrayList<>();
    public static void main(String[] args) {
       values.add("changed");       //     DOES COMPILE: values is a reference variable; you can call
                                    //                   methods on reference variables
       values = new ArrayList<>(); // DOES NOT COMPILE: we cannot reassign the final values to point to
}                                   //                   a different object
```

4.4.9 Static Initialization

- Static initializers look similar to instance initializers, but they add the `static` keyword to indicate they should be run when the class is first used:

```
private static final int NUM_SECONDS_PER_HOUR;
static {                                     // runs when class is first used
  int numSecondsPerMinute = 60;              // statements run and assign any static variables as needed
  int numMinutesPerHour = 60;
  NUM_SECONS_PER_HOUR =                      // the static initializer is the first assignment, therefore
    numSecondsPerMinute * numMinutesPerHour; // we can assign a value here to the final variable
}
```

- But:

```
private static int one;                  // COMPILES: it's initialized in the static initializer block
private static final int two;            // COMPILES: it's initialized in the static initializer block
private static final int three = 3;      // COMPILES: it's initialized in the declaration
private static final int four;           // DOES NOT COMPILE: final variable is never initialized (also not
                                         //                   inside the static initializer block

static {
    one = 1;                             // COMPILES: initializes static variable
    two = 2;                             // COMPILES: initializes final static variable for the first time
    three = 3;                           // DOES NOT COMPILE: has already been initialized in the
                                         //                   declaration; cannot reassign another value
                                         //                   to a final variable that has already been
                                         //                   initialized
    two = 4;                             // DOES NOT COMPILE: has already been initialized within the
                                         //                  static block; cannot reassign a second time
}
```

- Try to avoid static and instance initializers; use constructors instead, which makes code easier to read.

4.4.10 Static Imports

- Regular imports are for importing classes.
- Static imports are for importing static members of classes with the idea that you don't have to specify where each static method or variable comes from each time you use it.
- You can use wildcards to import static members or import a specific member.
- Example:

```
import java.util.List;
import static java.util.Arrays.asList;          // static import of asList member from the Arrays class
public class StaticImports {
    public static void main(String[] args) {
       List<String> list = asList("one", "two"); // instead of doing Arrays.asList("one", "two)
    }                                            // if you would create an asList method within your own
}                                                // class, that would take precedence
```

- The exam will try to trick you with misusing static imports:

```
import static java.util.Arrays;          // DOES NOT COMPILE: static imports are not for importing classes
import static java.util.Arrays.asList;   //         COMPILES: static import is for importing static members
static import java.util.Arrays.*;        // DOES NOT COMPILE: syntax is import static, not static import
public class BadStaticImports {
    public static void main(String[] args) {
       Arrays.asList("one");             // DOES NOT COMPILE: static member asList has been imported, but
    }                                    //                   the Arrays class has not been imported
}                                        //                   instead write: asList("one") or import Arrays
```

- Importing two classes with the same name gives a compilation error. This is also true for static imports:

```
import static statics.A.TYPE;
import static statics.B.TYPE;      // DOES NOT COMPILE
```

4.5 Passing Data Among Methods

- Java is a "pass-by-value" language, i.e. a copy of the variable is made and the method receives that copy.
- Assignments made in the method do not affect the caller.
- Some other languages use "pass-by-references", which does affect the caller.
- Example with primitive type:

```
public static void main(String[] args) {
    int num = 4;
    newNumber(num);
    System.out.println(num);        // prints 4 (remains unchanged by the newNumber method cal: no
}                                   // assginments are made to num from method newNumber within the main
                                    // method)

public static void newNumber(int num) {
    num = 8;                        // num parameter within method is set to 8, but this num parameter is
}                                   // not related to the num variable in the main method, they just have
                                    // the same name
```

- Example with reference type:

```
public static void main(String[] args) {
    String name = "Webby";
    speak(name);
    System.out.println(name);       // prints Webby (remains unchanged by the speak method call: no
}                                   // assignments are made to name from method speak in the main method)

public static void speak(String name) {
    name = "Sparky";                // name parameter within method is set to Sparky, but this name
}                                   // parameter is not related to the name variable in the main method,
                                    // they just have the same name
```

- But we can call methods on parameters which does affect the caller, e.g.:

```
public static void main(String[] args) {
    StringBuilder name = new StringBuilder();
    speak(name);
    System.out.println(name);       // Webby
}

public static void speak(StringBuilder s) {
    s.append("Webby");              // a method on the parameter is called; it doesn't reassign name to
}                                   // a different object; s is a copy of the variable name and both point
                                    // to the same StringBuilder, so changes to StringBuilder are
                                    // available in both references
```

- To get data back from a method a copy is made of the primitive or reference and returned from the method. This returned value can be stored in a variable by the caller or it can **be ignored (watch out for ignored values on the exam)**:

```
public class ReturningValues {
    public static main(String[] args) {
       int number = 1;                          // 1      initialized with 1
       String letters = "abc";                  // abc    initialized with abc
       number(number);                          // 1      calls numbers, which increases 1 by 1, but
                                                //        returned number is not reassigned to number, so
                                                //        it remains 1
       letters = letters(letters);              // abcd   calls letters, which adds d to letters and
                                                //        returned value is reassigned to letters, so abcd
       System.out.println(number + letters);   // 1abcd
    }

    public static int number(int number) {
       number++;
       return number;
    }

    public static String letters(String letters) {
       letters += "d";
       return letters;
    }
}
```

4.6 Overloading Methods

- Creating methods with the same name in the same class is called method overloading, i.e. the class has different method signatures with the same name but with different type parameters.
- Overloading also allows different numbers of parameters.
- We can overload:
    - By changing any parameter in the parameter list.
    - We can have a different type, more types, or the same types in a different order.
    - Access modifier and exception list are irrelevant to overloading .

  Examples:

```
public void fly(int numMiles) {}
public void fly(short numFeet) {}
public boolean fly() { return false; }
void fly(int numMiles, short numFeet) {}
public void fly(short numFeet, int numMiles) throws Exception {}
public int fly(short numFeet) {}        // DOES NOT COMPILE: only differs from original by return type;
                                        //                   parameter list is same as second example so
                                        //                   this is a duplicate method
public static void fly(int numMiles)    // DOES NOT COMPILE: parameter list is same as first example;
                                        //                   only difference is the specifier static so
                                        //                   this is a duplicate method
```

- To call an overloaded method you just call it with the right parameter(s). Java looks for matching types and calls the appropriate method.

4.6.1 Overloading and Varargs

- Example:

```
1. public fly(int[] lengths) {}
2. public fly(int... lengths) {}   // DOES NOT COMPILE: varargs are treated as an array; duplicate
                                   //                   method (same method signature)

Both methods (1) and (2) can be called by passing an array as follows:
fly(new int[] { 1, 2, 3});

But we can only call the varargs method (2) with stand-alone parameters:
fly(1, 2, 3);                      // only works when calling method 2, not when calling method 1
```

4.6.2 Autoboxing

- This means converting primitive types like `int` to its object type of `Integer`.
- When having both primitive `int` and object `Integer` version of a method, the most specific parameter is used:

```
1. public void fly(int numMiles) {}
2. public void fly(Integer numMiles) {}

fly(3)                  // will call 1, but if 1 isn’t present, it autoboxes int into Integer and call 2
fly(new Integer(3));    // will call 2, but if 2 isn’t present, it autoboxes Integer into int and call 1
```

4.6.3 Reference Types

- Java picks the most specific version of a method that it can:

```
public class ReferenceTypes {
    public void fly(String s) {     // first method
       System.out.println(“string “);
    }
    public void fly(Object o) {     // second method
       System.out.println(“object “);
    }
    public static void main(String[] args) {
       ReferenceTypes r = new ReferenceTypes();
       r.fly(“test”);               // passes a string, so calls first method accepting a string parameter
       r.fly(56);                   // passes an int, but there is no method with int parameter,
    }                               // so autoboxes int to Integer, but there is no method with Integer
}                                   // parameter, so calls the second method (Integer is an Object)
```

4.6.4 Primitives

- Primitives work like reference variables: Java tries to match with the most specific overloaded method:

```
public class Plane {
    public void fly(int i) {        // first method
       System.out.println(“int “);
    }
    public void fly(long l) {       // second method
       System.out.println(“long “);
    }
    public static void main(String[] args) {
       Plane p = new Plane();
       p.fly(123);                  // passes an int, so calls first method accepting an int parameter
                                    // if first method isn’t there, int is autoboxed into long and second
                                    // method gets called
       p.fly(123L);                 // passes a long, so calls second method accepting a long parameter
    }
}
```

- Note that Java can only accept wider types: an `int` can be passed to a method taking a `long` parameter (Java will convert it to a narrower type), but not vice versa without explicit casting to its narrower type, e.g.:

```
public class Plane {
    public void fly(int i) {
       System.out.println(“int “);
    }
    public static void main(String[] args) {
       Plane p = new Plane();
       p.fly(123L);         // DOES NOT COMPILE: cannot pass a long to a method taking an int parameter
       p.fly((int)123L);    //         COMPILES: explicit casting to int will work (potentially losing
                            //                   data, since a real long value might not fit into an int)
    }
}
```

4.6.5 Putting it all together

- When some of the types interact, Java rules focus on backward compatibility: autoboxing and varargs come last (since they were introduced in later versions of the language) when looking at overloaded methods:

Rule	Example of what will be chosen for glide(1, 2)
Exact match by type	`public String glide(int i, int j) {}`
Larger primitive type	`public String glide(long i, long j) {}`
Autoboxed type	`public String glide(Integer i, Integer j) {}`
Varargs	`public String glide(int... nums) {}`

- Java will only do one conversion:

```
public class TooManyConversions {
    public static void play(Long l) {}      // method 1
    public static void play(Long... l) {}   // method 2
    public static void main(String[] args) {
        play(4);        // DOES NOT COMPILE: there are no play methods accepting an int, long or Integer
                        //                   parameter; Java can autobox int 4 to a long or an Integer
                        //                   (first conversion), but it cannot autobox a second time to a
                        //                   Long type
        play(4L);       //          COMPILES: autoboxes long to Long (only one conversion needed) and calls
}                       //                   method 1
```

4.7 Creating Constructors

- Constructors are used when creating a new object, called *instantiation*: it creates a new instance of the class.
- It is called by writing the keyword `new` followed by the name of the class we want to instantiate.
- It is typically used to initialize instance variables.
- The `this` keyword tells Java you want to reference an instance variable.
- Most of the time the `this` keyword is optional, but it is mandatory when two variables have the same name, where one is the parameter name in the constructor and the other the instance variable; the most granular scope, which is the parameter, takes precedence in this case:

```
public class Bunny {
    private String color;
    public Bunny(String color) {
        this.color = color;         // here we need the this keyword to refer to the instance variable
    }
}
```

- But:

```
public class Bunny {
    private String color;
    public Bunny(String newColor) {
        color = newColor;           // no name clash here, so we don't have to use the this keyword
    }
}
```

- And beware on the exam for:

```
public class Bunny {
    private String color;
    public Bunny(String color) {
        color = this.color;         // is perfectly legal, but doesn't have the wanted result; it's
    }                               // backwards and doesn't change the value of the instance variable
}
```

4.7.1 Default Constructor

- If you don't include any constructors in a class, Java will create one for you without any parameters during the compile step (so it is only in the compiled class file). This is called the *default constructor* or default no-arguments constructor, which is the same as writing one yourself which doesn't have any parameters and an empty body.

- Having a private constructor in a class tells the compiler not to provide a default no-argument constructor. It also prevents other classes from instantiating the class. This is useful when a class only has static methods or the class wants to control all calls to create new instances of itself.

4.7.2 Overloading Constructors

- You can have multiple constructors in the same class as long as they have different method signatures.
- When overloading methods, the method name needs to match; with constructors, the name is always the same since it has to be the same as the name of the class, therefore constructors must have different parameters in order to be overloaded:

```
public class Hamster {
    private String color;
    private int weight;
    public Hamster(int weight) {  // 1st constructor
        this.weight = weight;
        color = "brown";
    }
    public Hamster(int weight,
        String color) {           // 2nd constructor; same name as 1st, different parameters, so overloaded
        this.weight = weight;
        this.color = color;
    }
}
```

- To avoid duplication (as in the above example) we can call the second constructor from the first, but how?

```
public Hamster(int weight) {
    Hamster(weight, "brown");     // DOES NOT COMPILE: a constructor can only be called using new before
                                  //                   the name of the constructor; they're not like
                                  //                   normal methods you can just call
}

public Hamster(int weight) {
    new Hamster(weight, "brown"); // COMPILES: but doesn't do what we want, it creates yet another Hamster
}

public Hamster(int weight) {
    this(weight, "brown");        // COMPILES: calls 2nd constructor without creating a new instance;
                                  //           using this calls another constructor on the same instance
}                                 //           of the class
```

- The `this` keyword to call another constructor has a special rule: it needs to be the first non-commented statement in the constructor:

```
public Hamster(int weight) {
    System.out.println("in constructor");
    // ready to call this         // comment is allowed
    this(weight, "brown");        // DOES NOT COMPILE: it's not the first statement
}
```

- One common technique is to have each constructor add one parameter until getting to the constructor that does all the work. This is called *constructor chaining*.

4.7.3 Final Fields

- Final instance variables must be assigned a value exactly once. This can be done in three ways:
    - In one line during declaration.
    - In instance initializer blocks.
    - In the constructor, since it is part of the initialization process, where it is allowed to assign final instance variables.

4.7.4 Order of Initialization

- The following four rules apply only if an object is instantiated. If a class is referred to without a new call, only rules 1 and 2 apply:
    1. If there is a superclass, initialize it first.
    2. Static variable declarations and static initializers in the order they appear in the file.
    3. Instance variable declarations and instance initializers in the order they appear in the file.
    4. The constructor.

- Example 1:

```
public class InitializationOrderSimple {    // RULE 1: does not apply, no superclass
   private String name = "Torchie";         // RULE 3: instance variable declarations and instance
   { System.out.println(name); }            //         initializers in the order they appear; prints out
                                            //         Torchie
   private static int COUNT = 0;            // RULE 2: static variable declaration
   static {                                 //         static initializer block in the order they appear
      System.out.println(COUNT);            //         prints out 0
   }                                        //
   static {                                 //
      COUNT += 10;                          //
      System.out.println(COUNT);            //         prints out 10
   }                                        //
   public InitializationOrderSimple() {     // RULE 4: the constructor comes last
      System.out.println("constructor");    //         prints out constructor
   }                                        //
}

public class CallInitializationOrderSimple {
   public static void main(String[] args) {
      InitializationOrderSimple init =
         new InitializationOrderSimple();
   }
}
```

Outputs:

```
0
10
Torchie
Constructor
```

- Example 2:

```
public class InitializationOrder {          // RULE 1: does not apply, no superclass
   private String name = "Torchie";         // RULE 3: instance variable declarations/instance initiali-
   { System.out.println(name); }            //         zers in the order they appear; prints out Torchie
   private static int COUNT = 0;            // RULE 2: static variable declaration
   static {                                 //         static initializer block in the order they appear
      System.out.println(COUNT);            //         prints out 0
   }                                        //
   {                                        //
      COUNT++;                              // RULE 3: the instance initializer increases COUNT by 1
      System.out.println(COUNT);            //         and prints out 1
   }                                        //
   public InitializationOrder() {           // RULE 4: the constructor comes last
      System.out.println("constructor");    //         prints out constructor
   }                                        //
   public static void main(String[] args) {
      System.out.println("construct");      // after RULE 1 and RULE 2 (the statics) the main method can
      new InitializationOrder ();           // run, printing construct
   }
}
```

Outputs:

```
0
construct
Torchie
1
constructor
```

- Example 3:

```
public class YetMoreInitializationOrder {   // RULE 1: does not apply, no superclass
   static { add(2); }                       // RULE 2: static variable declarations/instance initializer
   static void add(int num) {
      System.out.print(num + "");
   }
   YetmoreInitializationOrder() {           // RULE 4: the constructor comes last
      add(5);
   }
   static { add(4); }                       // RULE 2: static variable declarations/instance initializer
   { add(6); }                              // RULE 3: instance initializer
   static {                                 // RULE 2: static variable declarations/instance initializer
      new YetMoreInitializationOrder();
   }
   { add(8); }                              // RULE 3: instance initializer
   public static void main(String[] args) {
   }
}
```

Outputs:

```
2 4 6 8 5
```

4.8 Encapsulating Data

- Encapsulation means we set up the class so only methods in the class with the variables can refer to the instance variables. Callers are required to use these methods.
- With the getter (or accessor method) we read the value of an instance variable.
- With the setter (or mutator method) we can update the value of an instance variable.
- A caller cannot directly access the instance variables. In our setter methods we could do some validation (guard conditions), e.g. prevent setting a negative value for an `int` variable.
- So, the data (instance variables) are private, the getters and setters are public.
- Naming convention is the same as defined in *JavaBeans* (reusable software components), where an instance variable is called a *property*:

Rule	**Example**
Properties are private.	`private int numEggs;`
Getter methods begin with `is` or `get` if the property is a `boolean`.	`public boolean isHappy() { return happy; }`
Getter methods begin with `get` if the property is not a `boolean`.	`public int getNumEggs() { return numEggs; }`

Setter methods begin with `set`.	`public void setHappy(boolean happy) {`
	`   this.happy = happy`
	`}`

The method name must have a prefix of `set/get/is`, followed by the first letter of the property in uppercase, followed by the rest of the property name.	`public void setNumEggs(int num) {`
	`   numEggs = num`
	`}`

- The method parameter can be named anything you want. Only the method name and property name have naming conventions here.
- Examples of proper JavaBeans naming conventions:

```
private boolean playing;                          // CORRECT
private String name;                              // CORRECT
public boolean getPlaying() { return playing; } // CORRECT: for boolean get or is is allowed
public boolean isPlaying() { return playing; }  // CORRECT: for Boolean get or is is allowed
public String name() { return name; }             // INCORRECT: does not start with get
public void updateName(String n) { name = n; }  // INCORRECT: does not start with set
public void setname(String n) { name = n; }      // INCORRECT: name should begin with capital N
```

4.8.1 Creating Immutable Classes

- Encapsulating data prevents callers from making uncontrolled changes to your class.
- Another method is to make classes immutable so they cannot be changed at all:
  - They will always be the same.
  - Are easier to maintain.
  - Helps with performance by limiting the number of copies.
- To make a class immutable we omit setters and create constructors to set initial values.
- Immutable is only measured after the object is constructed, so initialization in constructor is fine.
- Immutable classes are allowed to have values, they just can't change after instantiation.
- When writing immutable classes, be careful about the return types:

```
public class NotImmutable {
    private StringBuilder builder;
    public NotImmutable(StringBuilder b) {
       builder = b;
    }
    public StringBuilder getBuilder() {
       return builder;
    }
}
```

```
public class TestNotImmutable {                         // here we pass the same StringBuilder all over
    public static void main(String args[]) {
       StringBuilder sb = new StringBuilder("initial");
       NotImmutable problem = new NotImmutable(sb);
       sb.append(" added");
       StringBuilder gotBuilder = problem.getBuilder(); // anyone calling the getter gets a reference to
                                                        // the sb created in the first line of the method
       gotBuilder.append(" more");
       System.out.println(problem.getBuilder());        // prints: initial added more
    }
}
```

- To make it immutable, a copy of the mutable object should be made (defensive copy):

```
public class Immutable {
    private StringBuilder builder;
    public Immutable(StringBuilder b) {                 // we create a new StringBuilder, which will not
       builder = new StringBuilder(b);                  // change after instantiation; Immutable no longer
    }                                                   // cares about the original StringBuilder passed
    public StringBuilder getBuilder() {                 // we create a new StringBuilder, which will not
       return new StringBuilder(builder);               // change; callers can change the StringBuilder
    }                                                   // original value without affecting Immutable
}

public class TestImmutable {
    public static void main(String args[]) {
       StringBuilder sb = new StringBuilder("initial"); // initial value of sb
       Immutable im = new Immutable(sb);
       sb.append(" added");                             // initial sb is changed, but does not affect
                                                        // Immutable
       StringBuilder gotBuilder = im.getBuilder();      // returns the Immutable object
       gotBuilder.append(" more");                      // gotBuilder is changed, but does not affect
                                                        // Immutable
       System.out.println(im.getBuilder());             // prints: initial
    }
}
```

- Another way to make it immutable would be to have the getter return an immutable object:

```
public class Immutable {
    ...
    public String getValue() {    // we don't have to return the same type as we are storing; String is
       return builder.toString(); // immutable
    }
}
```

4.9 Writing Simple Lambdas

- Functional programming is a way of writing code more declaratively:
    - You specify what you want to do rather than dealing with the state of objects.
    - You focus more on expressions than loops.
    - It uses lambda expressions to write code.
- A lambda is a block of code that gets passed around as if it were a variable; it's like an anonymous method.
- A lambda has parameters and a body.
- It doesn't have a name like a real method.
- Only the simplest lambda expressions are on the OCA exam.

4.9.1 Lambda Example

- Example:

```
public class Animal {
    private String species;
    private boolean canHop;
    private boolean canSwim;
    public Animal(String speciesName, boolean hopper, boolean swimmer) {
       species = speciesName; canHop = hopper; canSwim = swimmer;
    }
    public boolean canHop() { return canHop; }
    public boolean canSwim() { return canSwim; }
    public String toString() { return species; }
}
public interface CheckTrait {     // an interface specifies the methods that our class needs to implement
    boolean test(Animal a);
}
```

```
import java.util.ArrayList;
public class TraitFinder {
    public static void main(String[] args) {
       List<Animal> animals = new ArrayList<>();
       animals.add(new Animal(“fish”, false, true));
       animals.add(new Animal(“kangaroo”, true, false));
       animals.add(new Animal(“rabbit”, true, false));
       animals.add(new Animal(“turtle”, false, true));

       print(animals, a -> a.canHop());     // here we use lambda to find all animals that can hop
       print(animals, a -> a.canSwim());    // here we use lambda to find all animals that can swim
       print(animals, ! a.canSwim());       // here we use lambda to find all animals that cannot swim
                                            // NOTE: traditionally, without using lambda, we had to write
                                            // classes for each trait (CheckIfHopper, CheckIfSwimmer)
                                            // that implemented the CheckTrait interface
    }
    private static void print(List<Animal> animals, CheckTrait checker) {
       for (Animal animal : animals) {
          if (checker.test(animal)) {
             System.out.print(animal + “ ”);
          }
       }
       System.out.println();
    }
}
```

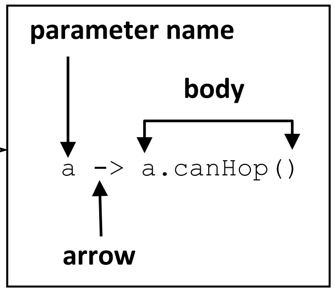
- The above example demonstrates the concept of deferred execution: the code is specified “now” but will run later; in this case later is when the `print()` method calls it.

4.9.2 Lambda Syntax

- The lambda in the example above (`a -> a.canHop`) means that Java should call a method with an `Animal` parameter that returns a `boolean` value that’s the result of `a.canHop()`.
- Java relies on context when figuring out what the lambda expressions mean:
    - In the above example we pass the lambda as a second parameter of the `print()` method.
    - This method expects a `CheckTrait` as the second parameter.
    - Java tries to map the lambda to that interface: `boolean test(Animal a);`
    - That interface’s method takes an `Animal`, so the lambda parameter has to be an `Animal`.
    - That interface’s method returns a `boolean`, therefore the lambda returns a `boolean`.
- Many parts of the lambda syntax are optional; the following are equivalent:

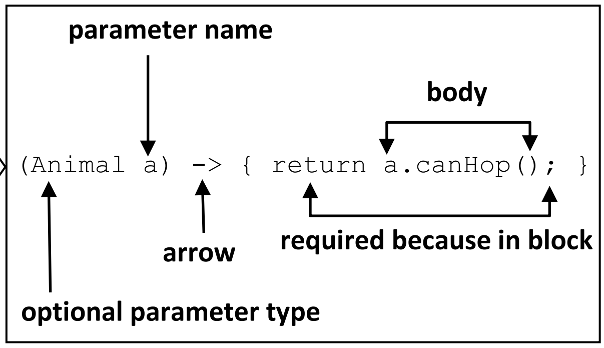
<u>Lambda syntax omitting optional parts:</u>

- Specifies a single parameter with the name `a`.
- Uses arrow operator to separate parameter and body.
- A body calling a single method and returns the result of that method.



<u>Lambda syntax including optional parts (more verbose):</u>

- Specifies a single parameter with the name `a` and stating the type is `Animal`.
- Uses arrow operator to separate parameter and body.
- A body that has one or more lines of code, including a semicolon and a return statement.



- Parentheses can only be omitted if there is a single parameter and its type is not explicitly stated. It doesn’t work when we have two or more statements. In other words: parentheses are *only* optional when there is one parameter and it doesn’t have a type declared.

- There isn't a rule that says you must use all defined parameters as seen in the below valid lambda examples:

```
print(() -> true);                                   // 0 parameters
print(a -> a.startsWith("test"));                    // 1 parameter
print((String a) -> a.startsWith("test"));           // 1 parameter
print((a, b) -> a.startsWith("test"));               // 2 parameters
print((String a, String b) -> a.startsWith("test")); // 2 parameters
```

- Examples:

```
print(a, b -> a.startsWith("test"));        // DOESN'T COMPILE: more than 1 parameter, needs parentheses
print(a -> {a.startsWith("test");});        // DOESN'T COMPILE: missing return keyword
print(a -> {return a.startsWith("test")}); // DOESN'T COMPILE: missing semicolon
```

- Lambdas are allowed to access variables (not on the OCA exam). They cannot access all variables, only instance and static variables and method parameters and local variables if these are not assigned new values.
- In the lambda expression we defined an argument list (which are local variables). Java doesn't allow us to redeclare a local variable, so:

```
(a, b) -> { int a = 0; return 5; }     // DOES NOT COMPILE: trying to redeclare a which is not allowed
(a, b) -> { int c = 0; return 5; }     // COMPILES: it uses a different variable name c
```

4.9.3 Predicates

- Lambdas work with interfaces that have only one method, so called functional interfaces.
- To prevent writing lots of interfaces, Java provides an interface from the package `java.util.function`:

```
public interface Predicate<T> {  // uses type T (generics) instead of a concrete class
    boolean test(T t);           // which means we don't need our own interface
}
```

  Referring to the earlier example with our print method we can pass it a `Predicate` with type `Animal` instead of using our own interface `CheckTrait`:

```
import java.util.ArrayList;
public class TraitFinder {
    public static void main(String[] args) {
       ...
       print(animals, a -> a.canHop());
    }

    private static void print(List<Animal> animals, Predicate<Animal>) {
       ...
    }
}
```

- For the exam you need to know that `ArrayList` declares a `removeIf()` method that takes a `Predicate`:

```
List<String> bunnies = new ArrayList<>();
bunnies.add("long ear");
bunnies.add("floppy");
bunnies.add("hoppy");
System.out.println(bunnies);                 // prints: [long ear, floppy, hoppy]
bunnies.removeIf(s -> s.charAt(0) != 'h');   // removeIf defines a Predicate taking a String and returning
                                             // a boolean; removes all elements from the list that don't
                                             // start with character 'h'
System.out.println(bunnies);                 // prints: [hoppy]
```

- For the OCA exam, you only need to know how to implement lambda expressions that use the `Predicate` interface.

4.10 Exam Essentials

- ***Be able to identify correct and incorrect method declarations:*** e.g. `public static void method(String... args) throws Exception {}`.
- ***Identify when a method or field is accessible:*** recognize when a method or field is accessed when the access modifier (`private`, `protected`, `public`, or default access) does not allow it.
- ***Recognize valid and invalid uses of static imports:*** static imports import static members, written as `import static`, **not** `static import`. Make sure they're importing static methods or variables rather than classnames.
- ***State the output of code involving methods:*** identify when to call static rather than instance methods based on whether the classname or object comes before the method. Recognize the correct overloaded method. Exact

matches are used first, followed by wider primitives, followed by autoboxing, followed by varargs. Assigning new values to method parameters does not change the caller, but calling methods on them does.

- ***Evaluate code involving constructors:*** constructors can call other constructors by calling `this()` as the first line of the constructor. Recognize when the default constructor is provided. Remember the order of initialization is the superclass, static variables/initializers, instance variables/initializers, and the constructor.
- ***Be able to recognize when a class is properly encapsulated:*** look for `private` instance variables and `public` getters and setters when identifying encapsulation.
- ***Write simple lambda expressions:*** look for the presence or absence of optional elements in lambda code. Parameter types are optional. Braces and the `return` keyword are optional when the body is a single statement. Parentheses are optional when only one parameter is specified and the type is implicit. The `Predicate` interface is commonly used with lambdas because it declares a single method called `test()`, which takes one parameter.

5 Class Design

5.1 OCA Exam objectives

1. Working with Inheritance:
   a. Describe inheritance and its benefits.
   b. Develop code that demonstrates the use of polymorphism; including overriding an object type versus reference type.
   c. Determine when casting is necessary.
   d. Use `super` and `this` to access objects and constructors.
   e. Use abstract classes and interfaces.

5.2 Introducing Class Inheritance

- You can define a class to inherit from an existing class.
- Inheritance is the process by which the new child subclass (descendent) automatically includes any `public` or `protected` primitives, objects, or methods defined in the parent (ancestor) class.
- Java only supports single inheritance: a class may only inherit from one direct parent class.
- An exception is: classes may implement multiple interfaces.
- Java does support multiple levels of inheritance: one class may extend another class, which in turn also extends another class (this can occur any number of times).
- Multiple inheritance can lead to complex and difficult to maintain code.
- To prevent a class to be extended, you can mark it with the `final` modifier (results in compiler error if you try to extend a `final` class).

5.2.1 Extending a Class

- To inherit from a parent class you use the `extends` keyword after the name of the subclass followed by the name of the parent class:

public or default access modifier

abstract or final keyword (optional)

class keyword (required)

class name

extends parent class (optional)

```
public abstract class ElephantSeal extends Seal {
      // Methods and variables defined here
}
```

- Java allows only one public class per file, so we create two files, each with its own public class:

```
public class Animal {
   private int age;
   public int getAge() {
      return age;
   }
   public void setAge(int age) {
      this.age = age;
   }
}

public class Lion extends Animal {      // Lion class extends from Animal class; Lion object is actually
   private void roar() {                // "bigger" than the Animal object: it includes all of the
      System.out.prinln("The "          // properties of the Animal object along with the Lion attributes
         + getAge()                     // we can access the public method from the parent class
         + " year old lion says: Roar!");
   }
}
```

- The `public` and default package-level class access modifiers are the only ones that can be applied to top-level classes within a Java file. The `protected` and `private` modifiers can only be applied to inner classes which are classes that are defined within other classes (out of scope of OCA).

- A Java file can have many classes but **at most** one public class. It may have no public class at all though.
- There can be **at most** one `public` class or interface in a Java file.
- Top-level interfaces can also be declared with the `public` or default modifiers.

5.2.2 Applying Class Access Modifiers

- For OCA you only need to know the `public` and default package-level class access modifiers. These can be applied to top-level classes within a Java file.
- The `protected` and `private` class access modifiers can only be applied to inner classes (classes defined within other classes). This is out of scope for OCA.
- A Java file can have many classes but **at most** one `public` class.
- There can be **at most** one `public` class or interface in a Java file.

5.2.3 Creating Java Objects

- All classes inherit from a single class, `java.lang.Object`, which is the only class that doesn't have any parent classes.
- When Java sees you define a class that doesn't extend another class, it immediately adds the syntax `extends java.lang.Object` to the class definition (at compile time).

5.2.4 Defining Constructors

- The first statement of every constructor is either a call to another constructor within the class, using `this()`, or a call to a constructor in the direct parent class, using `super()`.
- If a parent constructor takes arguments, the `super` constructor would also take arguments.
- The `super()` command may only be used as the first statement of the constructor:

```
public class Zoo {
    public Zoo() {
       System.out.prinln("Zoo created");
       super();         // DOES NOT COMPILE: super() should be the first statement
    }
}
```

- If the parent class has more than one constructor, the child class may use any valid parent constructor in its definition:

```
public class Animal {
    private int age;
    private String name;
    public Animal(int age, String name) {
       super();
       this.age = age;
       this.name = name;
    }
    public Animal(int age) {
       super();
       this.age = age;
       this.name = null;
    }
}

public class Gorilla extends Animal {
    public Gorilla(int age) {     // child constructor with one argument
       super(age, "Gorilla);      // calling parent constructor with two arguments
    }
    public Gorilla() {            // child constructor without arguments
       super(5);                  // calling parent constructor with one argument
    }
}
```

- Child constructors are not required to call matching parent constructors.
- Any valid parent constructor is acceptable as long as the appropriate input parameters to the parent constructor are provided.

5.2.4.1 Understanding Compiler Enhancements

- The Java compiler automatically inserts a call to the no-argument constructor `super()` if the first statement is not a call to the parent constructor.

- If the parent class does not have a no-argument constructor, you must create at least one constructor in your child class that explicitly calls a parent constructor via the `super()` command:

```
public class Mammal {
    public Mammal(int age) {
    }
}
public class Elephant extends Mammal { // DOES NOT COMPILE: parent class does not have constructor that
}                                     //                    takes no arguments

public class Elephant extends Mammal {
    public Elephant() {               // we define constructor that takes no arguments
    }                                 // DOES NOT COMPILE: compiler tries to insert super() as the first
                                      //                    statement, but there is no such constructor
}                                     //                    in the parent class
public class Elephant extends Mammal {
    public Elephant() {               // no-argument constructor (though its parent doesn't have a
       super(10);                     // no-argument constructor); explicit call to super constructor
    }                                 // of parent class taking one argument, which does compile
}
```

- **Watch out on the exam:** subclasses may define no-argument constructors even if their parent classes do not, provided the constructor of the child maps to a parent constructor via an explicit call of the `super()` command.

5.2.4.2 Reviewing Constructor Rules

- The first statement of every constructor is a call to another constructor within the class using `this()`, or a call to a constructor in the direct parent class using `super()`.
- The `super()` call may not be used after the first statement of the constructor.
- If no `super()` call is declared in a constructor, Java will insert a no-argument `super()` as the first statement of the constructor.
- If the parent doesn't have a no-argument constructor and the child doesn't define any constructors, the compiler will throw an error and try to insert a default no-argument constructor into the child class.
- If the parent doesn't have a no-argument constructor, the compiler requires an explicit call to a parent constructor in each child constructor.

5.2.4.3 Calling Constructors

The parent constructor is always executed before the child constructor. Watch out for this on the exam.

5.2.5 Calling Inherited Class Members

- Java classes may use any `public` or `protected` member of the parent class, including methods, primitives, or object references.
- If parent and child are part of the same package, the child class may also use any default members defined in the parent class.
- A child class may never access a private member of the parent class (not through direct reference).
- To reference a member in the parent class, you can call it directly:

```
class Fish {
    protected int size;
    private int age;
    public Fish(int age) {
       this.age = age;
    }
    public int getAge() {
       return age;
    }
}
public class Shark extends Fish {
    private int numberOfFins = 8;
    public Shark(int age) {
       super(age);
       this.size = 4;
    }
    pubic void displaySharkDetails() {
       System.out.print("Shark with age: " + getAge());         // public getter to access value in parent
       System.out.print(" and " + size + " meters long");       // protected member size
       System.out.print(" with " + numberOfFins + " fins");     // private numberOfFins of child class
    }
}
```

- You may also use `this` keyword to access members of the parent class that are accessible from the child class:

```
pubic void displaySharkDetails() {
    System.out.print("Shark with age: " + this.getAge());
    System.out.print(" and " + this.size + " meters long");
    System.out.print(" with " + this.numberOfFins + " fins");
}
```

- You may also use `super` keyword to access members of the parent class that are accessible from the child class:

```
pubic void displaySharkDetails() {
    System.out.print("Shark with age: " + super.getAge());
    System.out.print(" and " + super.size + " meters long");
    System.out.print(" with " + this.numberOfFins + " fins");
}
```

- We **cannot** access a member of the child class using the `super` keyword:

```
pubic void displaySharkDetails() {
    System.out.print("Shark with age: " + super.getAge());
    System.out.print(" and " + super.size + " meters long");
    System.out.print(" with " + super.numberOfFins + " fins"); // DOES NOT COMPILE: numberOfFins is a
}                                                              // member of current class, not of its
                                                               // parent
```

- The `this` and `super` keywords may be used for methods or variables defined in the parent class, but only `this` may be used for members defined in the current class.
- The `this` keyword and `this()` are unrelated. The `super` keyword and `super()` are also unrelated:
    - `super()` explicitly calls a parent constructor and may only be used in the first line of a constructor of a child class.
    - `super` is a keyword to reference a member defined in a parent class and may be used throughout the child classes.

5.2.6 Inheriting Methods

- Inheriting a class grants us access to the `public` and `protected` members of the parent class.
- It can also lead to collisions between methods defined in both the parent class and the subclass.

5.2.6.1 Overriding a Method

- Overriding a method means that you define a new method in the child class with the same signature (name and list of input parameters) and return type as the method in the parent class.
- You can still reference the parent version using the `super` keyword.
- The keywords `this` and `super` allow you to select between the current and parent version of the method, resp.
- Example:

```
public class Canine {
    public double getAverageWeight() {
       return 50;
    }
}
public class Wolf extends Canine {
    public double getAverageWeight() {  // overrides parent's getAverageWeight; we need super keyword here
       return super.getAverageWeight()  // to prevent recursive call: without super the method would call
          +20;                          // itself, causing infinite loop (since there is no termination
    }                                   // condition) resulting in a stack overflow error at runtime
    public static void main(String[] args) {
       System.out.println(new Canine().getAverageWeight());    // prints 50.0
       System.out.println(new Wolf().getAverageWeight());      // prints 70.0
    }
}
```

- The compiler performs the following checks when you override a nonprivate method:
    - The method in the child class:
        - Must have the same signature as the method in the parent class.
        - Must be at least as accessible or more accessible than the method in the parent class.
        - May not throw a checked exception that is new or broader than the class of any exception thrown in the parent class method.

    - If the method returns a value, it must be the same or a subclass of the method in the parent class (covariant return types).
- If two methods have the same name but different signatures, the methods are overloaded, not overridden.
- Both overloading and overriding involve redefining a method using the same name: an overloaded method will use a different signature than an overridden method.
- Examples:

```
public class InsufficientDataException extends Exception {}

public class Reptile {
    protected boolean hasLegs() throws InsufficientDataException {
       throw new InsufficientDataException();
    }
    protected double getWeight() throws Exception() {
       return 2;
    }
}

public class Snake extends Reptile {
    protected boolean hasLegs() { // does not throw an exception whereas parent does; this is OK since no
       return false;             // new exception is defined; child method may hide or eliminate a parent
    }                            // method's exception
    protected double getWeight()
       throws InsufficientDataException {   // parent throws Exception, child throws
          return 2;                         // InsufficientDataException; this is OK since the latter is
    }                                       // a subclass of Exception
}
```

```
public class InsufficientDataException extends Exception {}

public class Reptile {
    protected double getHeight() throws InsufficientDataException() {
       return 2;
    }
    protected int getLength() {
       return 10;
    }
}

public class Snake extends Reptile {
    protected double getHeight()
       throws Exception {  // DOES NOT COMPILE: parent throws InsufficientDataExcepion whereas child
          return 2;        //                   throws Exception; this is not OK since the latter is not a
    }                      //                   subclass of InsuffcientDataException
    protected int getLength()
       throws InsufficientDataException {   // DOES NOT COMPILE: parent doesn't throw an exception
          return 10;                        //                   whereas child does (child defines a new
    }                                       //                   exception); this is not OK
}
```

5.2.6.2 Redeclaring private Methods

- It is not possible to override a `private` method in a parent class since the parent method is not accessible from the child class.
- The child class can define its own version of the method though (but not an overridden version), i.e. a method with the same name and same or modified signature; this is however an independent method (unrelated to the parent). In this case none of the rules for overriding methods apply.

5.2.6.3 Hiding Static Methods

- A hidden method occurs when a child class defines a static method with the same name and signature as a static method defined in a parent class.
- Method hiding looks similar to method overriding: the rules for method overriding must be followed, but another rule also applies:
    - The method defined in the child class must be marked as `static` if it is marked as `static` in the parent class (method hiding). The method must not be marked as `static` in the child class if it is not marked `static` in the parent class (method overriding).
- Method hiding should be avoided.

- Examples:

```
public class Bear {
    public static void eat() {
        System.out.println("Bear is eating");
    }
}


public class Panda extends Bear {
    public static void eat() {    // method hiding (not overriding): parent and child are both static
        System.out.println("Panda bear is chewing");
    }
    public static void main(String[] args) {
        Panda.eat();
    }
}
```

```
public class Bear {
    public static void sneeze() {
        System.out.println("Bear is sneezing");
    }
    public void hibernate() {
        System.out.println("Bear is hibernating");
    }
}


public class Panda extends Bear {
    public void sneeze() {        // DOES NOT COMPILE: method is marked static in parent but not in child
        System.out.println("Panda bear sneezes quietly");
    }
    public static void hibernate() {    // DOES NOT COMPILE: is instance method in parent, but static
                                        //                   in child
        System.out.println("Panda bear is going to sleep");
    }
}
```

5.2.6.4 Overriding vs. Hiding Methods

- In overriding a method a child method replaces the parent method in calls defined in both the parent and child.
- Hidden methods only replace the parent methods in the calls defined in the child class.
- At runtime the child version of an overridden method is always executed for an instance regardless of whether the method call is defined in the parent or child class method: the parent method is never used unless an explicit call to the parent method is referenced using the syntax `super.method()`.
- At runtime the parent version of a hidden method is always executed if the call to the method is defined in the parent class.

5.2.6.5 Creating final methods

- Methods that are `final` cannot be overridden.
- By marking methods as `final` you forbid a child class from overriding these methods. This applies both to method overriding and method hiding. In other words: you cannot hide a static method in a parent class if it is marked as `final`:

```
public class Bird {
    public final boolean hasFeathers() {
        return true;
    }
}

public class Penguin extends Bird {
    public final boolean hasFeathers() {    // DOES NOT COMPILE: since method is marked final in parent
        return false;
    }
}
```

- You'd mark a method as `final` when you're defining a parent class and want to guarantee certain behavior of a method in the parent class, regardless of which child is invoking the method.

5.2.7 Inheriting Variables

Java doesn't allow variables to be overridden, but they can be hidden.

5.2.7.1 Hiding Variables

- When hiding a variable, you define a variable with the same name as a variable in a parent class.
- When referencing the variable from within the parent class, the variable defined in the parent class is used.
- When referencing the variable from within a child class, the variable defined in the child class is used.
- You can reference the parent value of the variable with an explicit use of the `super` keyword.
- There is no notion of overriding a member variable. The rules are the same regardless of whether the variable is an instance variable or a static variable.
- Hiding non-private variables is bad practice and can lead to confusion/bugs:

```
public class Animal {
    public int length = 2;
}

public class Jellyfish extends Animal {
    public int length = 5;
    public static void main(String[] args) {
        Jellyfish jellyfish = new Jellyfish();    // we create the same type of object twice (Jellyfish),
        Animal animal = new Jellyfish();          // but the reference to the object determines which value
                                                  // is seen as output (Polymorphism)
        System.out.println(jellyfish.length);     // prints 5
        System.out.pringln(animal.length);        // prints 2
    }
}
```

- Hiding private variables is considered less problematic because the child class does not have access to them in the parent class.

5.3 Creating Abstract Classes

- An *abstract class* is a class that is marked with the `abstract` keyword and cannot be instantiated.
- An *abstract method* is a method marked with the `abstract` keyword defined in an `abstract` class, for which no implementation is provided in the class in which it is declared.
- By defining an `abstract` class as a parent class with `abstract` methods we prevent users from instantiating this class and force the child classes to implement the `abstract` methods.
- Example:

```
public abstract class Animal {                     // class declared abstract, since it has abstract mehod
    protected int age;
    public void eat() {                            // an abstract class can have nonabstract methods with an
        System.out.println("Animal is eating");    // implementation
    }
    public abstract String getName();              // abstract method terminated with semicolon (without
}                                                  // implementation); if a class has one or more abstract
                                                   // methods, the class needs to be abstract as well
public class Swan extends Animal {                 // nonabstract class
    public String getName() {                      // needs to implement the abstract method from its parent
        return "Swan";                             // class Animal; the method has the same name and
    }                                              // signature
}
```

5.3.1 Defining an Abstract Class

- An `abstract` class may include nonabstract methods and variables. In other words: you can still define a method with a body implementation in an `abstract` class, you just can't mark the method as `abstract`.
- An `abstract` class is not required to include any `abstract` methods.
- An `abstract` method may only be defined in an `abstract` class. The following is wrong (watch out for this on the exam):

```
public class Chicken {              // nonabstract class
    public abstract void peck();   // DOES NOT COMPILE: abstract method in a nonabstract class not allowed
}
```

- Also watch out for `abstract` methods which have an implementation defined, which is not allowed:

```
public abstract class Turtle {      // abstract class
    public abstract void swim() {} // DOES NOT COMPILE: abstract method cannot have empty implementation
    public abstract int getAge() { // DOES NOT COMPILE: abstract class cannot have an implementation
        return 10;
    }
}
```

- An abstract class cannot be marked as final: it is a class that must be extended by another class to be instantiated, whereas a final class can't be extended by another class:

```
public final abstract class Tortoise { }      // DOES NOT COMPILE: abstract class cannot be final
```

- An abstract method may also not be marked as final (same reason as above): once marked as final, a method cannot be overridden in a subclass:

```
public abstract class Goat {
    public abstract final void chew();  // DOES NOT COMPILE: abstract method cannot be final; it needs to
}                                       // be overridden (implemented) by the child class extending Goat
```

- A method may not be marked both abstract and private: an abstract method needs to be implemented by the subclass, but if the method is private, it wouldn't be accessible by the subclass:

```
public abstract class Whale {
    private abstract void sing(); // DOES NOT COMPILE: method cannot be both abstract and private
}
public class HumpbackWhale extends Whale {
    private void sing() {
       System.out.println("Humpback whale is singing");
    }
}
```

- With abstract methods, the rules for overriding methods must be followed, therefore the following is wrong:

```
public abstract class Whale {
    protected abstract void sing();                      // COMPILES: method is now protected
}
public class HumpbackWhale extends Whale {
    private void sing() {                                // DOES NOT COMPILE: subclass cannot reduce visi-
       System.out.println("Humpback whale is singing"); //                     bility of the parent class
    }
}
```

5.3.2 Creating a Concrete class

- An abstract class cannot be instantiated, therefore the following is wrong:

```
public abstract class Eel {
    public static void main(String[] args) {
       final Eel eel = new Eel();       // DOES NOT COMPILE: cannot instantiate an abstract class
    }
}
```

- An abstract class becomes useful when it is extended by a concrete subclass.
- A *concrete class* is the first nonabstract subclass that extends an abstract class and is required to implement all inherited abstract methods.
- On the exam check that a concrete class implements all of the required methods.
- Example:

```
public abstract class Animal {
    public abstract String getName();
}
public class Bird extends Animal {      // DOES NOT COMPILE: first concrete class must implement all
}                                       //                     abstract methods

public class Flamingo extends Bird {
    public String getName() {           // A second concrete subclass implements the abstract method from
       return "Flamingo";               // its abstract parent class Animal, but the first concrete
    }                                   // subclass should have done so already
}
```

5.3.3 Extending an Abstract Class

- We can extend an abstract class with another abstract class:

```
public abstract class Animal {
    public abstract String getName();
}
public class Walrus extends Animal {          // DOES NOT COMPILE: first concrete class must implement all
}                                             abstract methods
public abstract class Eagle extends Animal { // this is OK, since Eagle is also defined as abstract and
                                              // doesn't have to implement the abstract methods of its
}                                             // parent abstract class
```

- A concrete class that extends an abstract class must implement all inherited abstract methods (from all it's parent abstract classes), unless an intermediate abstract class already provided an implementation:

```
public abstract class Animal {
   public abstract String getName();
   public abstract int getAge();
}
public abstract class BigCat extends Animal {
   public abstract void roar();
   public int getAge() {          // it is OK to implement abstract method from an abstract parent in an
      return 5;                   // intermediate abstract class; in this case the first concrete
   }                              // subclass doesn’t have to implement it any more, but it can still
}                                 // override it though: if an intermediate class provides an implemen-
                                  // tation for an abstract method, that method is inherited by sub-
                                  // classes as a concrete method, not as an abstract one
public class Lion extends BigCat {
   public String getName() {      // abstract method must be implemented in first concrete class, since
      return “Lion”;              // it hasn’t been implemented in any intermediate abstract parent
   }                              // class (in this case it isn’t implemented in BigCat abstract parent)
   public void roar() {           // abstract method must be implemented in first concrete class, since
      System.out.println(“ROAR!”);// it hasn’t been implemented in any intermediate abstract parent
   }                              // class (in this case it isn’t implemented in BigCat abstract parent)
}
```

- Abstract Class Definition Rules:
  - Abstract classes:
    - Cannot be instantiated directly.
    - May be defined with any number, including zero, of abstract and nonabstract methods.
    - May not be marked as `private`, `protected`, or `final`.
  - An abstract class that extends another abstract class inherits all of its abstract methods as its own abstract methods.
  - The first concrete class that extends an abstract class must provide an implementation for all the inherited abstract methods.
- Abstract Method Definition Rules:
  - Abstract methods:
    - May only be defined in abstract classes.
    - May not be declared `private` or `final`.
    - Must not provide a method body/implementation in the abstract class for which it is declared.
  - Implementing an abstract method in a subclass follows the same rules for overriding a method: the name and signature must be the same and visibility of the method in the subclass must be at least as accessible as the method in the parent class.

5.4 Implementing Interfaces

- An *interface* is an abstract data type defining a list of `abstract public` methods that any class implementing that interface must provide.
- An interface can also include a list of constant variables and default methods (the latter since Java 8).
- An interface is defined with the `interface` keyword:

public or default access modifier

abstract or final keyword (assumed)

interface keyword (required)

interface name

```
public abstract interface CanBurrow {
      public static final int MINIMUM_DEPTH = 2;
      public abstract int getMaximumDepth();
}
```

public static final keywords assumed

public abstract keywords assumed

- A class invokes the interface by using the `implements` keyword in its class definition:

class name

implements keyword (required)

interface name

```
public class FieldMouse implements CanBurrow {
      public int getMaximumDepth() {
            return 10;
      }
}
```

signature matches interface method

- Java allows implementing any number of interfaces; multiple interfaces are separated by a comma:

```
public class Elephant implements WalkOnFourLegs, HasTrunk, Herbivore { }
```

5.4.1 Defining an Interface

- An interface is a special kind of abstract class sharing many of the same properties and rules as an abstract class:
  - Interfaces cannot be instantiated directly.
  - An interface is not required to have any methods.
  - An interface may not be marked as `final`.
  - All top-level interfaces are assumed to have `public` or default access. They are assumed `abstract` whether this keyword is used or not. Therefore making a method `private`, `protected` or `final` results in a compilation error.
  - All nondefault methods in an interface are assumed to have the modifiers `abstract` and `public` in their definition. Therefore, making a method as `private`, `protected`, or `final` results in a compilation error.
- Examples of illegal interface definitions:

```
public class TestClass {
  public static void main(String[] args) {
   WalksOnTwoLegs example = new WalksOnTwoLegs();   // DOES NOT COMPILE: cannot instantiate an interface
  }
}

public final interface WalksOnTwoLegs { }    // DOES NOT COMPILE: interface cannot be final

private final interface CanCrawl {     // DOES NOT COMPILE: interface cannot be final (conflicts with
                                       //                   abstract keyword); interface cannot be private
                                       //                   (conflicts with required public or default
                                       //                   access for interfaces)
  private void dig(int depth);         // DOES NOT COMPILE: interface methods are assumed to be public
  protected abstract double depth();   // DOES NOT COMPILE: interface methods are assumed to be public
  public final void surface();         // DOES NOT COMPILE: interface method cannot be final (conflicts
                                       //                   with the assumption that they are abstract)
}
```

5.4.2 Inheriting an Interface

- Two inheritance rules are:
  - An interface that extends another interface, as well as an abstract class that `implements` an interface, inherits all of the abstract methods as its own abstract methods.
  - The first concrete class that implements an interface, or `extends` an abstract class that `implements` an interface, must provide an implementation for all of the inherited abstract methods.
- An interface may be extended using the `extends` keyword: the new child interface inherits all the abstract methods of the parent interface. Any class that implements the `Seal` interface below must provide an implementation for all methods in the parent interfaces, i.e. `getTailLength()` and `getNumberOfWhiskers()`:

```
public interface HasTail {
  public int getTailLength();
}
public interface HasWhiskers {
  public int getNumberOfWhiskers();
}
public interface Seal extends HasTail, HasWhiskers { }
```

- An abstract class implementing an interface is treated in the same way as an interface extending another interface: the abstract class inherits the abstract methods of the interface but is not required to implement them.
- The first concrete class to extend the abstract class must implement all the inherited abstract methods of the interface:

```
public interface HasTail {
  public int getTailLength();
}
public interface HasWhiskers {
  public int getNumberOfWhiskers();
}
public abstract class HarborSeal implements HasTail, HasWhiskers { }
public class LeopardSeal
  implements HasTail, HasWhiskers {   // DOES NOT COMPILE: first concrete class must implement all the
}                                     //                   inherited abstract methods of the interfaces
                                      //                   that it implements
```

5.4.2.1 Classes, Interfaces, and Keywords

- A class can implement and interface, it cannot extend an interface.
- An interface can extend another interface, it cannot implement another interface.
- An interface cannot extend a class.
- Examples of wrong definitions:

```
public interface CanRun { }
public class Cheetah extends CanRun { }      // DOES NOT COMPILE: a class cannot extend an interface, it
                                             //                   can only implement it
public class Hyena { }
public interface HasFur extends Hyena { }    // DOES NOT COMPILE: an interface cannot extend a class, it
                                             //                   can only extend another interface
```

5.4.2.2 Abstract Methods and Multiple Inheritance

- A class can inherit from two interfaces that contain the same abstract method.
- If two abstract interface methods have identical behavior, i.e. the same method signature, creating a class that implements one of the two methods automatically implements the second method. The methods are compatible (duplicates):

```
public interface Herbivore {
  public void eatPlants();
}
public interface Omnivore {
  public void eatPlants();
  public void eatMeat();
}
public class Bear implements Herbivore, Omnivore {
  public void eatMeat() {
    System.out.println("Eating meat");
  }
  public void eatPlants() {
    System.out.println("Eating plants");
  }
}
```

- If two abstract methods have a different signature (same method name, different parameters), this is considered as method overloading; it doesn't matter if the return type of the two methods are the same or different:

```
public interface Herbivore {
  public void eatPlants(int quantity);
}
public interface Omnivore {
  public void eatPlants();
}
public class Bear implements Herbivore, Omnivore {      // implements both interfaces, therefore must
  public void eatPlants(int quantity) {                 // provide implementation of both versions of
    System.out.println("Eating plants " + quantity);    // eatPlants(int quantity) and eatPlants()
  }
  public void eatPlants() {
    System.out.println("Eating plants");
  }
}
```

- If the method name and input parameters are the same but the return types are different between the two methods, the class or interface attempting to inherit both interfaces will not compile:

```
public interface Herbivore {
  public int eatPlants();
}
public interface Omnivore {
  public void eatPlants();
}
public class Bear
  implements Herbivore, Omnivore {           // DOES NOT COMPILE: class defines two methods with the same
                                             // name and input parameters but different return types
  public int eatPlants() {                   // DOES NOT COMPILE
    System.out.println("Eating plants: 10");
    return 10;
  }
  public void eatPlants() {                  // DOES NOT COMPILE
    System.out.println("Eating plants");
  }
}
```

- If you define an interface or abstract class that inherits from two conflicting interfaces, you will also get a compilation error:

```
public interface Herbivore {
  public int eatPlants();
}
public interface Omnivore {
  public void eatPlants();
}
public interface Supervore extends Herbivore, Omnivore { }             // DOES NOT COMPILE
public abstract class AbstractBear implements Herbivore, Omnivore { }  // DOES NOT COMPILE
```

5.4.3 Abstract Methods and Multiple Inheritance

- Two interface variables rules:
    - Interface variables are assumed to be `public`, `static` and `final`. Therefore marking a variable as `private`, `protected` or `abstract` will result in a compilation error.
    - The value of an interface variable must be set when it is declared since it is marked as `final`.
- Interface variables are essentially constant variables defined on the interface level. Since they are `static` they can be accessed without an instance of the interface.
- The naming convention is the same as for constant variables: uppercase letters.
- The following definitions are wrong:

```
public interface canDig {
  private int MAXIMUM_DEPTH = 100;                // DOES NOT COMPILE: cannot be private
  protected abstract boolean UNDERWATER = false;  // DOES NOT COMPILE: cannot be protected and abstract
  public static String TYPE;                      // DOES NOT COMPILE: value should be set on declaration
}
```

5.4.4 Default Interface Methods (since Java 8)

- A *default method* is a method defined within an interface with the `default` keyword in which a method body is provided.
- It defines an abstract method with a default implementation: classes have the option to override the default method if they need to, but they are not required to do so.
- Default methods help with code development and backward compatibility.
- Example:

```
public interface IsWarmBlooded {
  boolean hasScales();                       // abstract method that needs to be implemented by the first
                                             // concrete class that implements this interface
  public default double getTemperature() {   // a default method that can be overridden by the class
    return 10.0;                             // implementing this interface, but doesn't have to be
  }                                          // overridden
}
```

- Default interface method rules:
    - A default method may only be declared within an interface and not within a class or abstract class.
    - A default method must be marked with the `default` keyword. If a method is marked as `default`, it must provide a method body.

- A default method is not assumed to be `static`, `final`, or `abstract`, as it may be used or overridden by a class that implements the interface.
- Like all methods in an interface, a default method is assumed to be `public` and will not compile if marked as `private` or `protected`.

- Example of wrong default method definitions:

```
public interface Carnivore {
  public default void eatMeat();        // DOES NOT COMPILE: default method should provide a
                                        //                   body/implementation
  public int getRequiredFoodAmount() { // DOES NOT COMPILE: not marked with default keyword, so no body/
    return 13;                          //                   implementation allowed
  }
}
```

- Default methods require an instance of the class implementing the interface to be invoked.
- When an interface extends another interface that contains a default method, it may ignore the default method, in which case the default implementation for the method will be used.
- The interface may also override the definition of the default method using the standard rules for method overriding, such as not limiting the accessibility of the method and using covariant returns.
- The interface may redeclare the method as abstract, requiring classes that implement the new interface to explicitly provide a method body.

5.4.4.1 *Default Methods and Multiple Inheritance*

- Java doesn't support multiple inheritance, so the following fails to compile:

```
public interface Walk {
  public default int getSpeed() {
    return 5;
  }
}
public interface Run {
  public default int getSpeed() {
    return 10;
  }
}
public class Cat implements Walk, Run {        // DOES NOT COMPILE:
  public static void main(String[] args) {     //   a concrete class can implement multiple interfaces, but
    System.out.println(new Cat().getSpeed()); //   if both interfaces have the same default methods (same
  }                                            //   name and signature), this will lead to a compilation
}                                              //   error; if this would not result in a compilation error,
                                               //   Java would allow multiple inheritance
```

- If the subclass overrides the duplicate default methods from the above example, the code would compile again; the ambiguity would be removed:

```
public class Cat implements Walk, Run {
  public int getSpeed() {          // a class that implements or inherits two duplicate default methods
    return 1;                      // is forced to implement a new version of the method
  }
  public static void main(String[] args) {
    System.out.println(new Cat().getSpeed());
  }
}
```

5.4.5 Static Interface Methods

- Java 8 now includes support for static methods within interfaces. The are defined with the `static` keyword.
- A static method defined in an interface is not inherited in any classes that implement the interface.
- Two static interface method rules:
  - Like all methods in an interface, a static method is assumed to be `public` and will not compile if marked as `private` or `protected`.
  - To reference the static method, a reference to the name of the interface must be used.

- Example:

```
public interface Hop {
  static int getJumpHeight() {           // this works just like a static method as defined in a class
    return 8;
  }
}

public class Bunny implements Hop {
  public void printDetails() {
    System.out.println(getJumpHeight());   // DOES NOT COMPILE: static interface methods are not inherited
                                           //                   by the class implementing the interface;
                                           //                   we need to add an explicit reference to
  }                                        //                   the name of the interface
}

public class Bunny implements Hop {
  public void printDetails() {
    System.out.println(Hop.getJumpHeight());
  }
}
```

- A class that implements two interfaces containing static methods with the same signature will still compile, because the static methods are not inherited by the subclass and must be accessed with a reference to the interface name.

5.5 Understanding Polymorphism

- Polymorphism is the property of an object to take on many different forms, in other words: a Java object may be accessed using a reference with the same type as the object, a reference that is a superclass of the object, or a reference that defines an interface the object implements, either directly or through a superclass.
- A cast is not required if the object is being reassigned to a super type or interface of the object.
- Only one object, `Lemur`, is created and referenced in the following example. The ability of an instance `Lemur` to be passed as an instance of an interface it implements, `HasTail`, as well as an instance of one of its superclasses, `Primate`, is the nature of polymorphism:

```
public class Bunny Primate {
  public boolean hasHair() {
    return true;
  }
}
public interface HasTail {
  public Boolean isTailStriped();
}
public class Lemur extends Primate implements HasTail {
  public Boolean isTailStriped() {
    return false;
  }
  public int age = 10;
  public static void main(String[] args) {
    Lemur lemur = new Lemur();
    System.out.println(lemur.age);                // prints 10

    HasTail hasTail = lemur;                      // polymorphism: we can assign to an interface
    System.out.println(hasTail.isTailStriped()); // prints false

    Primate primate = lemur;                      // polymorphism: we can assign to a parent
    System.out.println(primate.hasHair());        // prints true
  }
}
```

- Once the object has been assigned a new reference type, only the methods and variables available to that reference type are callable on the object without explicit cast:

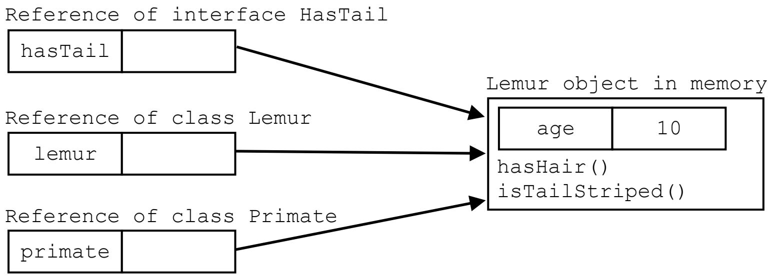
```
HasTail hasTail = lemur;                        // reference hasTail has direct access only to methods
                                                // defined with the HasTail interface
System.out.println(hasTail.age);                // DOES NOT COMPILE: reference type is HasTail, not Lemur

Primate primate = lemur;                        // reference primate has direct access only to methods
                                                // defined in the Primate class
System.out.println(primate.isTailStriped());   // DOES NOT COMPILE: reference type is Primate, not HasTail
```

5.5.1 Object vs. Reference

- All objects are accessed by reference; you never have direct access to the object itself. The object itself doesn't change:

```
Lemur lemur = new Lemur();          // Lemur is an object in memory and the object itself doesn't change
Object lemurAsObject = lemur;       // what has changed, is our ability to access methods within the Lemur
                                    // class with the lemurAsObject reference
```

- The following two rules apply:
  - The type of the object determines which properties exist within the object in memory.
  - The type of the reference to the object determines which methods and variables are accessible to the Java program.
- Changing a reference of an object to a new reference type may give you access to new properties of the object, but those properties existed before the reference change occurred. Depending on the type of the reference, we may only have access to certain methods, e.g. `hasTail` reference has access to method `isTailStriped()` but doesn't have access to the variable `age` defined in the `Lemur` class:



It is possible to reclaim access to the variable `age` by explicitly casting the `hasTail` reference to a reference of type `Lemur`.

5.5.2 Casting Objects

- When changing the reference type, you can lose access to more specific methods defined in the subclass that still exist within the object (as seen in the above figure). Access can be reclaimed by casting the object back to the specific subclass it came from:

```
Primate primate = lemur;
Lemur lemur2 = primate;             // DOES NOT COMPILE: cannot assign primate to a lemur without explicit
                                    //                   cast
Lemur lemur3 = (Lemur) primate;     // COMPILES if we use an explicit cast to a subclass of the object
System.out.println(lemur3.age);     //          Primate, which gives us access to all available methods of
                                    //          the Lemur class
```

- Basic rules for casting variables:
  - Casting an object from a subclass to a superclass doesn't require an explicit cast.
  - Casting an object from a superclass to a subclass requires and explicit cast.
  - The compiler will not allow casts to unrelated types. The exam may try to trick you here.
  - Even when the code compiles without issue, an exception may be thrown at runtime if the object being cast is not actually an instance of that class.
- Examples of incorrect casting with regards to third and fourth rule:

```
public class Bird { }
public class Fish {
  public static void main(String[] args) {
    Fish fish = new Fish();
    Bird bird = (Bird) fish;        // DOES NOT COMPILE: classes Fish and Bird are not related through any
  }                                 //                   class hierarchy that the compiler is aware of
}

public class Rodent { }
public class Capybara extends Rodent {
  public static void main(String[] args) {
    Rodent rodent = new Rodent();
    Capybara capybara = (Capybara) rodent; // throws ClassCastException at runtime: the object being
  }                                        // referenced is not an instance of the Capybara class; the
}                                          // object being created is not related to the Capybara class in
                                           // any way (Capybara is a Rodent, but Rodent is not a Capybara)
```

- When casting is involved on the exam, be sure to remember what the instance of the object actually is. Then focus on whether the compiler will allow the object to be reference with or without explicit casts.

5.5.3 Virtual Methods

- The most important feature of polymorphism is the support of virtual methods.
- A *virtual method* is a method in which the specific implementation is not determined until runtime.
- All non-final, non-static, and non-primitive Java methods are considered virtual methods: any of them can be overridden at runtime.
- If you call a method on an object that overrides a method, you get the overridden method, even if the call to the method is on a parent reference or within the parent class.
- Example:

```
public class Bird {
  public String getName() {
    return "Unknown";
  }
  public void displayInformation() {
    System.out.println("The bird name is: " + getName());
  }
}
public class Peacock extends Bird {
  public String getName() {
    return "Peacock";
  }
  public static void main(String[] args) {
    Bird bird = new Peacock();
    bird.displayInformation();    // prints: The bird name is: Peacock → the value of the getName()
  }                               // method at runtime in the displayInformation() method is replaced
}                                 // with the value of the implementation in the subclass Peacock
```

- Even though the parent class `Bird` defines its own version of `getName()` in the above example and doesn't know anything about the `Peacock` class during compile-time, at runtime the instance uses the overridden version of the method, as defined on the instance of the object. This is emphasized by using a reference to the `Bird` class in the `main()` method, although the result would have been the same if a reference to a `Peacock` was used.

5.5.4 Polymorphic Parameters

- Polymorphism has the ability to pass instances of a subclass or interface to a method (also known as polymorphic parameters), e.g. you can define a method that takes an instance of an interface as a parameter. Any class that implements that interface can be passed to the method. Since you're casting from a subtype to a supertype, an explicit cast is not required:

```
public class Reptile {
  public String getName() {
    return "Reptile";
  }
}
public class Alligator extends Reptile {  // Alligator is a subclass of Reptile; therefore we can pass
  public String getName() {               // it without any problem to the feed method which takes a
    return "Alligator";                   // Reptile as parameter
  }
}
public class Crocodile extends Reptile {  // Crocodile is a subclass of Reptile; therefore we can pass
  public String getName() {               // it without any problem to the feed method which takes a
    return "Crocodile";                   // Reptile as parameter
  }
}
public class Zooworker {
  public static void feed(Reptile reptile) {             // we can pass it a Reptile or any subclass of
    System.out.println("Feeding: " + reptile.getName()); // Reptile (Alligator and Crocodile), but we
  }                                                      // cannot pass an unrelated class such as Rodent
                                                         // or Capybara or java.lang.Object, which would
                                                         // result in a compilation error
  public static void main(String[] args) {
    feed(new Alligator());        // prints: Feeding: Alligator
    feed(new Crocodile());        // prints: Feeding: Crocodile
    feed(new Reptile());          // prints: Feeding: Reptile
  }
}
```

- It is considered good practice to use the superclass or interface type as input parameters to a method whenever possible. This is good for code reusability. A good example of this is to pass the interface `java.util.List` instead of the class `java.util.ArrayList()` or `java.util.Vector()`.

5.5.5 Polymorphism and Method Overriding

- The last three rules of method overriding are of importance for polymorphism:
  - An overridden method must be at least as accessible as the method it is overriding. If this wouldn't be the case, we would create an ambiguity problem.
  - A subclass cannot declare an overridden method with a new or broader exception than in the superclass, since the method may be accessed using a reference to the superclass.
  - Overridden methods must use covariant return types: if an object is cast to a superclass reference and the overridden method is called, the return type must be compatible with the return type of the parent method. If the return type in the child is too broad, it will result in an inherent cast exception when accessed through the superclass reference. E.g. if the return type of a method is `Double` in the parent class and is overridden in a subclass with a method that returns `Number`, a superclass of `Double`, then the subclass method would be allowed to return any valid `Number`, including `Integer`, another subclass of `Number`.

5.6 Exam Essentials

- ***Be able to write code that extends other classes:*** a class extending another class inherits all `public` and `protected` methods and variables. The first line of every constructor is a call to another constructor within the class using `this()` or a call to a constructor of the parent class using the `super()` call. If parent doesn't have a no-argument constructor, an explicit call to the parent constructor must be provided. Parent methods and instance variables can be accessed explicitly using the `super` keyword. All classes extend `java.lang.Object` (directly or via superclass).
- ***Understand the rules for method overriding:*** a method must have the same signature, be at least as accessible as the parent method, must not declare any new or broader exceptions, and must use covariant return types.
- ***Understand the rules for hiding methods and variables:*** when a static method is created in a subclass, it is referred to as method hiding. Variable hiding is when a variable name is reused in a subclass. For method hiding, the use of `static` in the method declaration must be the same between the parent and child class.
- ***Recognize the difference between method overriding and method overloading:*** both involve creating a new method with the same name as an existing method. When the method signature is the same, it is method overriding. When the method signature is different (taking different inputs), it is method overloading.
- ***Be able to write code that creates and extends abstract classes:*** abstract classes cannot be instantiated and require a concrete subclass to be accessed. They can include any number, including zero, of abstract and nonabstract methods. Abstract methods follow all the method override rules and may only be defined within abstract classes. The first concrete subclass of an abstract class must implement all the inherited methods. Abstract classes and methods may not be marked `final` or `private`.
- ***Be able to write code that creates, extends, and implements interfaces:*** only abstract methods and constant `static final` variables are allowed in interfaces. You can also define default and `static` methods with method bodies in interfaces. All members of an interface are assumed to be `public`. Methods are assumed to be `abstract` if not explicitly marked as default or `static`. An interface that extends another interface inherits all its abstract methods. An interface cannot extend a class, a class cannot extend an interface. Classes may implement any number of interfaces.
- ***Be able to write code that uses default and static interface methods:*** a default method allows a developer to add a new method to an interface used in existing implementations, without forcing other developers using the interface to recompile their code. A developer using the interface may override the default method or use the provided one. A static method in an interface follows the same rules for a static method in a class.
- ***Understand polymorphism:*** a Java object may take on a variety of forms partly depending on the reference used to access the object. Overridden methods will be replaced everywhere where they are used. Hidden methods and variables will only be replaced in the classes and subclasses that they are defined in. It is common to rely on polymorphic parameters when creating method definitions.

- ***Recognize valid reference casting:*** an instance can be automatically cast to a superclass or interface reference without an explicit cast. An explicit cast is required if the reference is being narrowed to a subclass of the object. Casting to unrelated types is not allowed. Recognize when casting results in compiler-time casting errors or in runtime errors that throw a `ClassCastException`.

6 Exceptions

6.1 OCA Exam objectives

1. Handling Exceptions:
   a. Differentiate among checked exceptions, unchecked exceptions and Error.
   b. Create a `try-catch` block and determine how exceptions alter normal program flow.
   c. Describe the advantages of `Exception` handling.
   d. Create and invoke a method that throws an exception.
   e. Recognize common exception classes (such as `NullPointerException`, `ArithmeticException`, `ArrayIndexOutOfBoundsException`, `ClassCastException`).

6.2 Understanding Exceptions

A program can fail for many reasons, some of which are caused by coding mistakes, others caused by reasons which are completely beyond your control, like an Internet connection not being available. In this case you can try to deal with the situation.

6.2.1 The Role of Exceptions

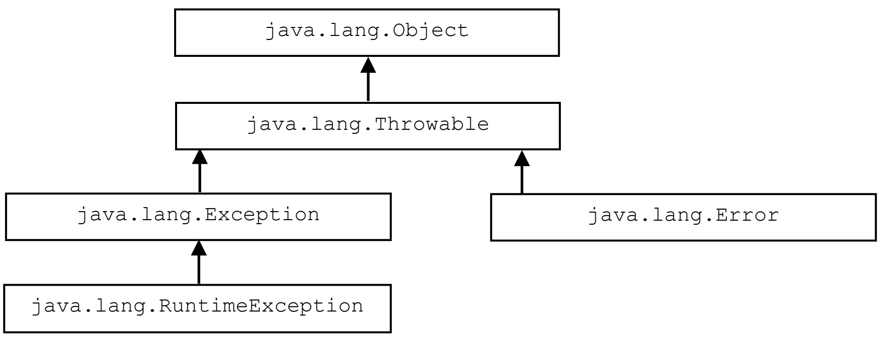
- The *happy path* is when nothing goes wrong (no exceptions have to be dealt with).
- If an exception occurs in a method, the method can handle that exception or make it the caller's responsibility.
- The following throws an exception which is not dealt with:

```
public class Zoo {
  public static void main(String[] args) {
    System.out.println(args[0]);
    System.out.println(args[1]);
  }
}

java Zoo Zoo                                  // when compiled and called with one argument
ZooException in thread "main"                 // throws exception since the args only contains one
java.lang.ArrayIndexOutOfBoundsException: 1   // element Zoo at index 0; index 1 is not allowed
at Zoo.main(Zoo.java:4)
```

- In more advanced programs you need to deal with failures in accessing files, networks, and outside services.
- Exceptions alter the program flow.
- In some cases code can return a return code instead of throwing an exception; this is common in searching algorithms where `-1` is returned if no match is found, which is expected behavior for searches. In general this kind of behavior should be avoided and exceptions should be used.
- An exception forces the program to deal with them or end with the exception if left unhandled; a return code could be accidentally ignored and cause problems later in the program.

6.2.2 Understanding Exception Types

- An exception is an event that alters program flow. Java has a `Throwable` superclass for all objects that represent these events. Not all of them have the word exception in their classname:



- Error means something went horribly wrong that your program should not attempt to recover from it.
- A *runtime exception* (or `unchecked exception`) is defined as the `RuntimeException` class and its subclasses, which tend to be unexpected but not necessarily fatal, like accessing an invalid array index.

- A *checked exception* includes `Exception` and all subclasses that do not extend `RuntimeException`, which tend to be more anticipated, like reading a file that doesn't exist. It forces programmers to do something to show the exception was thought about.
- Java knows the *handle or declare rule*: checked exceptions need either be handled or declared in the method signature:

```
void fall() throws Exception {     // declares that it might throw an exception using the keyword throws
  throw new Exception();           // we throw an exception using the throw keyword
}
```

- Runtime exceptions, like a `NullPointerException`, can occur in any method. Runtime methods don't have to be declared; if you would have to declare them, every single method would be cluttered.

6.2.3 Throwing an Exception

- The OCA exam is limited to exceptions that someone else has created (mostly provided by Java).
- On the exam you will see two types of code that result in an exception:
  1. Questions about exceptions can be hidden in questions that appear to be about something else:

```
String[] animals = new String[0];
System.out.println(animals[0]);// throws ArrayIndexOutOfBoundsException
```

  2. Explicitly request Java to `throw` an exception; `throw` tells Java you want some other part of the code to deal with the exception:

```
throw new Exception();
throw new Exception("Ow! I fell.");          // we can pass a String parameter with a message
throw new RuntimeException();
throw new RuntimeException("Ow! I fell.");   // we can pass a String parameter with a message
```

- The following rules are important:

Type	How to recognize	Okay for program to catch?	Is program required to handle or declare?
Runtime exception	Subclass of `RuntimeException`	Yes	No
Checked exception	Subclass of `Exception` but not subclass of `RuntimeException`	Yes	Yes
Error	Subclass of `Error`	No	No

6.3 Using a *try* Statement

- Java uses a try statement to separate the logic that might throw an exception from the logic to handle that exception:

The try keyword.

If an exception is thrown in a try statement, the catch clauses attempt to catch it.

Curly braces are required.

```
try {
   // The try block is also referred to as protected code
} catch (exception_type identifier) {
   // exception handler
}
```

The identifier refers to the caught exception object.

The catch keyword.

The type of exception you are trying to catch.

- The code in the `try` block is run normally.
- If any of the statements throw an exception that can be caught by the exception type listed in the `catch` block, the `try` block stops running and execution goes to the `catch` statement.
- If none of the statements in the `try` block throw an exception that can be caught, the `catch` clause is not run.
- Note that block and clause are used interchangeably (they mean the same).

- Some invalid try statements:

```
try                      // DOES NOT COMPILE: curly braces are required even if there is only one
  fall();                //                   statement inside the code blocks
catch
  System.out.println(“get up”);

try {                    // COMPILES: this is the correct syntax
  fall();
} catch {
  System.out.println(“get up”);

}

try {                    // DOES NOT COMPILE: try block doesn’t have anything after it; needs a catch or
  fall();                //                   finally block (see below)
}
```

6.3.1 Adding a *finally* Block

- The `try` statement also lets you run code at the end with a `finally` clause regardless of whether an exception is thrown:

A finally block can only appear as part of a try statement.

```
try {
   // protected code
} catch (exception_type identifier) {
   // exception handler
} finally {
   // finally block
}
```

The finally keyword.

The finally block always executes whether or not an exception occurs in the try block.

- There are two paths through code with both `catch` and a `finally`:
  - An exception is thrown: the `finally` block is run after the `catch` block.
  - No exception is thrown: the `finally` block is run after the `try` block completes.
- On the OCA exam a `try` statement must have a `catch` and/or `finally`. Having both is fine. Having neither is a problem. The `try-with-resources` (which allows neither `catch` nor `finally`) is only for the OCP exam.
- The exam will try to trick you with missing clauses or clauses in the wrong order:

```
try {                               // DOES NOT COMPILE: catch and finally block in wrong order
  fall();
} finally {
  System.out.println(“all better”);
} catch (Exception e) {
  System.out.println(“get up”);
}

try {                               // DOES NOT COMPILE: either catch or finally block must be present
  fall();
}

try {
  fall();
} finally {                         // COMPILES: catch is not required if finally is present
  System.out.println(“all better”);
}
```

- Most examples on the OCA exam with `finally` look like this:

```
String s = “”;
try {
  s += “t”;
} catch (Exception e) {
  s += “c”;
} finally {
  s += “f”;
}
s += “a”;
System.out.println(s);    // prints tfa since no exception is thrown finally is executed after try
                          // then code after finally is executed
```

- There is one exception where the `finally` block isn't executed: when `System.exit(0)` is called in the `try` or `catch` block, `finally` isn't executed; the program stops immediately.

6.3.2 Catching Various Types of Exceptions

- The OCA exam can define basic exceptions to show you the hierarchy. You must be able to recognize if it is a checked or an unchecked exception and you must then determine if any of the exceptions are subclasses of the others:

```
class AnimalsOutForAWalk extends RuntimeExcetion { }    // unchecked exception
class ExhibitClosed extends RuntimeException { }        // unchecked exception
class ExhibitClosedForLunch extends ExhibitClosed { }   // unchecked exception

public void visitPorcupine() {
  try {
   seeAnimal();                           // if no exception thrown, nothing is printed
  } catch (AnimalsOutForAWalk e) {        // first catch block, if animal is out for a walk, only this
   System.out.println("try back later"); // catch block is thrown
  } catch (ExhibitClosed e) {             // second catch block, if exhibit is closed, only this catch
   System.out.println("not today");      // block is thrown
  }
}
```

- The order of defining the `catch` blocks is important: Java looks at the order in which they appear. If it is impossible for one of the `catch` blocks to be executed, a compiler error about unreachable code occurs, which happens when a superclass is caught before a subclass:

```
public void visitMonkeys() {
  try {
   seeAnimal();
  } catch (ExhibitClosedForLunch e) {     // subclass exception (more specific); if thrown, this block is
   System.out.println("try back later"); // executed
  } catch (ExhibitClosed e) {             // superclass exception (less specific); if thrown, this block
   System.out.println("not today");      // is executed
  }
}

public void visitMonkeys() {
  try {
   seeAnimal();                           // if more specific ExhibitClosedForLunch is thrown, the first
  } catch (ExhibitClosed e) {             // catch block runs and there's no way for the second catch
   System.out.println("not today");      // block to ever run
  } catch (ExhibitClosedForLunch e) {     // → unreachable catch block
   System.out.println("try back later"); //
  }
}

public void visitSnakes() {
  seeAnimal();
} catch (RuntimeException e) {
   System.out.println("runtime exception");
} catch (ExhibitClosedException e) {      // DOES NOT COMPILE: if ExhibitClosed (which is a
   System.out.println("not today");      //                    RuntimeException) is thrown, the first
} catch (Exception e) {                   //                    catch block is executed; there is no way
   System.out.pringln("exception");      //                    to get to the second catch block
}
```

6.3.3 Throwing a Second Exception

- A `catch` or `finally` block can have any valid Java code in it, including another `try` statement.
- The following example shows that only the last exception to be thrown matters:

```
try {
  throw new RuntimeException();  // throws RuntimeException
} catch (RuntimeException e) {   // which is caught by first catch block
  throw new RuntimeException();  // which in turn throws another RuntimeException
} finally {                      // but finally block always runs
  throw new Exception();         // which throws an Exception; if there was not a finally block, the
}                                // second RuntimeException would be thrown, but this one is forgotten
                                 // since the finally block runs
```

- We often see `try/catch` inside a `finally` block to make sure it doesn't mask the exception from the `catch` block:

```
public String exceptions() {
  String result = "";
  String v = null;
  try {
    try {
       result += "before ";
       v.length();                    // throws NullPointerException since reference v is null
       result += "after ";            // therefore this line is never reached
    } catch (NullPointerException e) {
       result += "catch ";            // the NullPointer is caught and "catch " is appended to result
       throw new RuntimeException();  // then a new RuntimeException is thrown
    } finally {                       // but finally block is always executed
       result += "finally ";          // thus appending "finally " to result
       throw new Exception();         // and a new Exception is thrown
    }
  } catch (Exception e) {             // which is caught
    result += "done";                 // thus "done" is appended to result
  }
  return result;                      // returns "before catch finally done"
}
```

6.4 Recognizing Common Exception Types

For OCA you need to recognize three types of exceptions: runtime exceptions, checked exceptions, and errors. You need to recognize whether it's thrown by the JVM or a programmer.

6.4.1 Runtime Exceptions

- These extend `RuntimeException`.
- They don't have to be handled or declared.
- They can be thrown by the programmer or the JVM.

6.4.1.1 ArithmeticException

- Thrown by the JVM when code attempts to divide by zero.
- Example:

```
int answer = 11 / 0;

Exception in  thread "main" java.lang.ArithmeticException: / by zero
```

6.4.1.2 ArrayIndexOutOfBoundsException

- Thrown by the JVM when code uses an illegal index to access an array.
- Example:

```
int[] countsOfMoose = new int[3];
System.out.println(countsOfMoose[-1]);

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: -1  // tells what index was invalid

int total = 0;
int[] countsOfMoose = new int[3];
for (int i = 0; i <= countsOfMoose.length; i++) // should be < instead of <=
  total += countsOfMoose[i]

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3
```

6.4.1.3 ClassCastException

- Thrown by the JVM when an attempt is made to cast an object to a subclass of which it is not an instance.
- Example:

```
String type = "moose";
Integer number = (Integer) type; // DOES NOT COMPILE: Integer is not a subclass of String
System.out.println(countsOfMoose[-1]);

Object obj = type;
Integer number = (Integer) obj;   // cast fails at runtime; casting from Object to Integer is OK, the
                                  // compiler can't determine there's a String in that Object

Exception in thread "main" java.lang.ClassCastException: java.lang.String cannot be cast to
java.lang.Integer
```

6.4.1.4 IllegalArgumentException

- Thrown by the programmer to indicate that a method has been passed an illegal or inappropriate argument.
- Throwing this exception is a way for your program to protect itself.

- Example: instead of using the guard condition we can throw an `IllegalArgumentException` to tell the caller that something is wrong:

```
public void setNumberEggs(int numberEggs) {  // setter
  if (numberEggs >= 0) // guard condition
    this.numberEggs = numberEggs;
}

public void setNumberEggs(int numberEggs) {
  if (numberEggs < 0)
    throw new IllegalArgumentException("# eggs must not be negative");
  this.numberEggs = numberEggs;
}
```

6.4.1.5 NullPointerException

- Thrown by the JVM when there is a `null` reference where an object is required.
- Instance variables and methods must be called on non-null reference. If the reference is `null`, a `NullPointerExcetpion` will be thrown.
- Example:

```
String name;
public void printLength() throws NullPointerException {
  System.out.prinln(name.length());
}

Exception in thread "main" java.lang.NullPointerException
```

6.4.1.6 NumberFormatException

- Thrown by the programmer when an attempt is made to convert a string to a numeric type but the string doesn't have an appropriate format.
- Java provides methods to convert strings to numbers. If an invalid value is passed, a `NumberFormatException` is thrown, which is a subclass of `IllegalArgumentException`.
- Example:

```
Integer.parseInt("abc");

Exception in thread "main" java.lang.NumberFormatException: For input string: "abc"
```

6.4.2 Checked Exceptions

- These extend `Exception`, not `RuntimeException`;
- They **must** be handled or declared.
- They can be thrown by the programmer or the JVM.

6.4.2.1 FileNotFoundException

- Thrown programmatically when code tries to reference a file that does not exist.
- Is a subclass of `IOException`.

6.4.2.2 IOException

- Thrown programmatically when there's a problem reading or writing a file.

6.4.3 Errors

- These extend `Error`.
- They should **not** be handled or declared.
- They are thrown by the JVM.

6.4.3.1 ExceptionInInitializerError

- Thrown by the JVM when a static initializer throws an exception and doesn't handle it.
- If one of the static initializers throws an exception, Java can't start using the class.

- Example:

```
static {
  int[] countsOfMoose = new int[3];
  int num = countsOfMoose[-1];
}
public static void main(String[] args) { }
Exception in thread "main" java.lang.ExceptionInInitializerError  // gives information that something
Caused by: java.lang.ArrayIndexOutOfBoundsException: -1           // went wrong in static block, but also
                                                                  // the original cause of the problem;
                                                                  // Java fails to load the whole class
```

6.4.3.2 *StackOverflowError*

- Thrown by the JVM when a method calls itself too many times (infinite recursion).
- The stack runs out of room and overflows (`StackOverflowError`).
- Example:

```
public static void doNotCodeThis(int num) {
  doNotCodeThis(1);     // at least Java eventually throws an error, which is better than running into an
}                       // infinite loop, where Java just uses all the CPU

Exception in thread "main" java.lang.StackOverflowError
```

6.4.3.3 *NoClassDefFoundError*

- Thrown by the JVM when a class that the code uses is available at compile time but not runtime.
- This error won't show up in the code on the exam, you just need to know that it is an error.

6.5 Calling Methods That Throw Exceptions

- When you're calling a method that throws an exception, the rules are the same as within a method.

```
class NoMoreCarrotsException extends Exception {}
public class Bunny {
  public static void main(String[] args) {
    eatCarrot();         // DOES NOT COMPILE: NoMoreCarrotsException is a checked exception which must be
  }                      //                   handled or declared: 2 solutions for main method below:
  private static void eatCarrot()          // eatCarrot doesn't throw an exception, but declares that it
    throws NoMoreCarrotsException {}       // could throw a NoMoreCarrotsException; therefore we need to
}                                          // declare or handle it in the caller

public static void main(String[] args)
  throws NoMoreCarrotsException {          // declare exception
  eatCarrot();
}

public static void main(String[] args) {
  try {
    eatCarrot();
  } catch (NoMoreCarrotsException e) {     // handle exception
    System.out.print("sad rabbit");
  }
}
```

- The compiler is still on the lookout for unreachable code: declaring an unused exception isn't considered unreachable code; it gives the method the option to change the implementation to throw that exception in the future:

```
public void bad() {
  try {
    eatCarrot();
  } catch (NoMoreCarrotsException e) {     // DOES NOT COMPILE: Java knows that eatCarrot() can't throw a
    System.out.println("sad rabbit");      //                   checked exception (it is not declared in
  }                                        //                   the method), therefore this catch block
}                                          //                   can never be reached
public void good() throws NoMoreCarrotsException {
  eatCarrot();
}
private static void eatCarrot() {}
```

6.5.1 Subclasses

- When a class overrides a method from a superclass or implements a method from an interface, it's not allowed to add new checked exceptions to the method signature:

```
class CanNotHopException extends Exception {}
class Hopper {
  public void hop() {}
}
class Bunny extends Hopper {
  public void hop()             // DOES NOT COMPILE: hop() isn't allowed to throw any checked exceptions
    throws CanNotHopException {} // because the superclass Hopper doesn't declare any
}
```

- A subclass is allowed to declare fewer exceptions than the superclass or interface. This is legal because the callers are already handling them:

```
class Hopper {
  public void hop()
    throws CanNotHopException {}
}
class Bunny extends Hopper {
  public void hop() {} // allowed to leave out throws, since it's handled in the superclass Hopper
}
```

- A subclass not declaring an exception is similar to a method declaring it throws an exception that it never actually throws. This is legal.
- Similarly, a class is allowed to declare a subclass of an exception type. The superclass or interface has already taken care of a broader type:

```
class Hopper {
  public void hop()
    throws Exception {}
}
class Bunny extends Hopper {
  public void hop()
    throws CanNotHopException {}  // we can throw a subclass of exception; the broader type is already
}                                 // handled in the superclass; it could throw Exception directly, throw a
                                  // more specific type like CanNotHopException or throw nothing at all;
                                  // this applies to checked exceptions
```

- The following is legal:

```
class Hopper {
  public void hop() {}
}
class Bunny extends Hopper {
  public void hop()
    throws IllegalStateException {}// even though the superclass doesn't throw any exception, we can
}                                  // throw a runtime exception here; the declaration is redundant;
                                   // methods are free to throw any runtime exceptions they want without
                                   // mentioning them in the method declaration
```

6.5.2 Printing an Exception

- There are three ways to print an exception:
  1. Let Java print it out.
  2. Print just the message.
  3. Print where the stack trace comes from which is usually most helpful showing where the exception occurred in each method it passed through.
- Examples:

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

  } catch (Exception e) {
    System.out.println(e);                  // 1: java.lang.RuntimeException: cannot hop
    System.out.println(e.getMessage());     // 2: cannot hop
    System.out.println(e.printStackTrace()); // 3: java.lang.RuntimeException: cannot hop
                                            //    at trycatch.Handling.hop(Handling.java:15)
                                            //    at trycatch.Handling.main(Handling.main:7)
  }
}
private static void hop() {
  throw new RuntimeException("cannot hop");
}
```

- Checked exceptions are required to be handled or declared. They can be caught but nothing has necessarily to be done with them, so they "go away" (exceptions are swallowed); this is bad behavior, which can lead to problems further on in the code:

```
public static void main(String[] args) {
  String textInFile = null;
  try {
    readInFile();
  } catch (IOException e) {
    // ignore exception
  }
  System.out.println(textInFile.replace(" ", ""));  // results in NullPointerException, but it's not
}                                                   // clear there was an IOException earlier, since the
                                                    // exception was handled, but ignored; it's better
                                                    // to print a stacktrace or log a message and
                                                    // consider if it is wise to continue at all
private static void readInFile() throws IOException {
  throw new IOException();
}
```

6.6 Exam Essentials

- ***Differentiate between checked and unchecked exceptions:*** unchecked exceptions (runtime exceptions) are subclasses of `java.lang.RuntimeException`. All other subclasses of `java.lang.Exception` are checked exceptions.
- ***Understand the flow of a try statement:*** a `try` statement must have a `catch` or a `finally` block. Multiple `catch` blocks are allowed, provided no superclass exception type appears in an earlier `catch` block than its subclasses. The `finally` block runs last regardless of whether an exception is thrown.
- ***Identify whether an exception is thrown by the programmer or the JVM:*** `IllegalArgumentException` and `NumberFormatException` are commonly thrown by the programmer. Most of the other runtime exceptions are typically thrown by the JVM.
- ***Declare methods that declare exceptions:*** the `throws` keyword is used in a method declaration to indicate an exception might be thrown. When overriding a method, the method is allowed to throw fewer exceptions than the original version.
- ***Recognize when to use throw versus throws:*** the `throw` keyword is used when you actually want to throw an exception, e.g. `throw new RuntimeException()`. The `throws` keyword is used in a method declaration.

Appendix B – Study Tips

Taking the test

Reviewing Common Compiler Issues

Common tips to determine if Code Compiles:

- Keep an eye out for all reserved words. [Chapter 1]
- Verify brackets – `{}` – and parentheses – `()` – are being used correctly. [Chapter 1]
- Verify `new` is used appropriately for creating objects. [Chapter 1]
- Ignore all line indentation especially with `if-then` statemnets that do not use brackets `{}`. [Chapter 2]
- Make sure operators use compatible data types, such as the logical complement operator (`!`) only applied to boolean values, and arithmetic operators (`+`, `-`, `++`, `--`) only applied to numeric values. [Chapter 2]
- For any numeric operators, check for automatic numeric promotion and order or operation when evaluating an expression. [Chapter 2]
- Verify `switch` statements use acceptable data types. [Chapter 2]
- Remember `==` is not the same as `equals()`. [Chapter 3]
- String values are immutable. [Chapter 3]
- Non-void methods must return a value that matches or is a subclass of the return type of the method. [Chapter 4]
- If two classes are involved, make sure access modifiers allow proper access of variables and methods. [Chapter 4]
- Nonstatic methods and variables require an object instance to access. [Chapter 4]
- If a class is missing a default no-argument constructor or the provided constructors do not explicitly call `super()`, assume the compiler will automatically insert them. [Chapter 5]
- Make sure abstract methods do not define an implementation, and likewise concrete methods always define an implementation. [Chapter 5]
- You implement an interface and extend a class. [Chapter 5]
- A class can be cast to a reference of any superclass it inherits from or interface it implements. [Chapter 5]
- Checked exceptions must be caught; unchecked exceptions may be caught but do not need to be. [Chapter 6]
- `try` blocks require a `catch` and/or `finally` block for the OCA exam. [Chapter 6]