Java SE8 OCA

Contents

Java SE8 OCA

Session 1:
Java Building Blocks

Class Structure

- File name *.java e.g. MyClass.java
- Probably contains:

```
public class MyClass {
   //Some code
}
```


Class Members

- Classes can contain fields and methods:

```
public class MyClass {
   String name;
   public String getName(){
      return name;
   }
   public void setName(String newName){
      name = newName;
   }
}
```


Comments

- Single-line comment:

```
// comment finishes at the end of the line
```

- Multiple-line comment:

```
/* comment finishes
*  with asterisk slash
*/
```

- Javadoc comment:

```
/** Javadoc processes annotations like:
*  @author Jane Smith
*/
```


main() Method

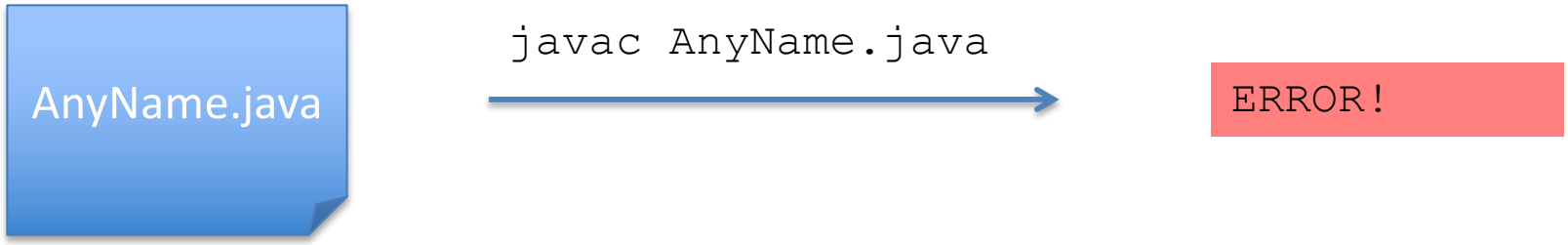
```
public class MyClass {
   public static void main(String[] args) {
      System.out.println(args[0]);
   }
}
```

Note: Modifiers may be in any order:

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


Files and 'public' Classes

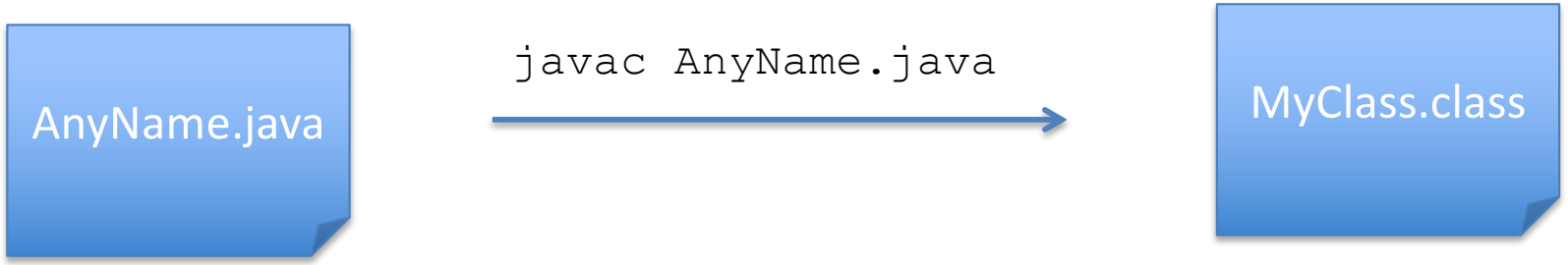

```
public class MyClass {
    //Some code
}
```




```
public class MyClass {
    //Some code
}
```


Files and ‘no modifier’ Classes


```
class MyClass {
    //Some code
}
```




```
class MyClass {
    //Some code
}
```


Compiling and Running Java Code

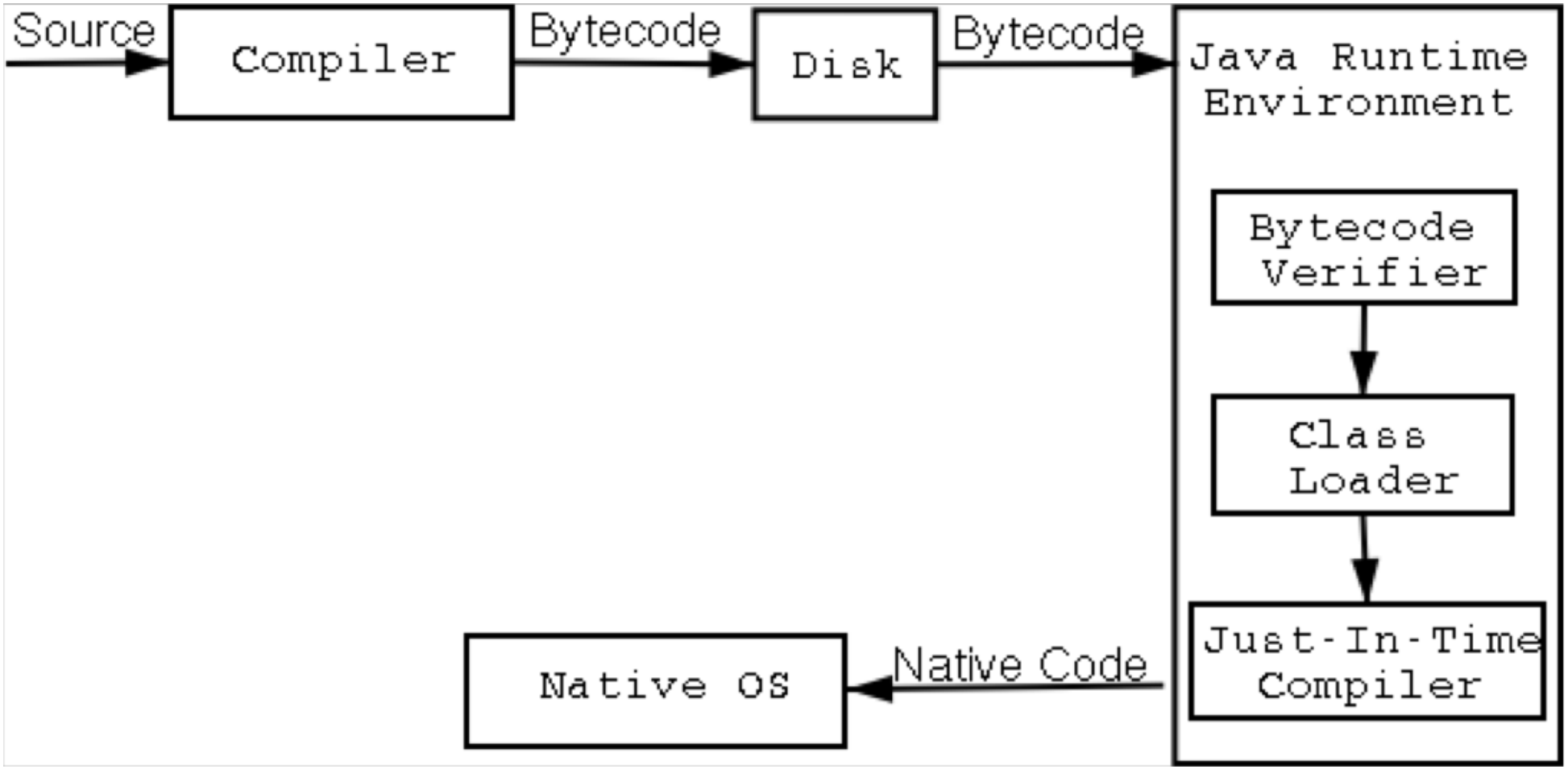
```
javac MyClass.java
```

- Should result in MyClass.class

```
java MyClass
```

- Should run MyClass.class and produce output to the console

Compilation



javac Command

```
c:\> javac -help
Usage: javac <options> <source files>
where possible options include:
    -g                         Generate all debugging info
    -nowarn                    Generate no warnings
    -verbose                   Output compiler activity messages
    -classpath <path>          Specify where to find user class files
    (or -cp)                   and annotation processors
    -d <directory>             Specify where to place generated class files
    -s <directory>             Specify where to putgenerated source files
    -version                   Version information
    -help                      Print a synopsis of standard options
    -Werror                    Terminate compilation if warnings occur
    @<filename>                Read options and filenames from file

Many more advanced options available
```


java Command

```
c:\> java -help
Usage: java [-options] class [args...]
           (to execute a class)
   or  java [-options] -jar jarfile [args...]
           (to execute a jar file)

 -cp <class search path of directories and zip/jar files>
    -classpath <class search path of directories and zip/jar files>
                  A : separated list of directories, JAR archives,
                  and ZIP archives to search for class files.

 -D<name>=<value>
                  set a system property

 -verbose:[class|gc|jni]
                  enable verbose output

-version      print product version and exit
```


Run as Background Process

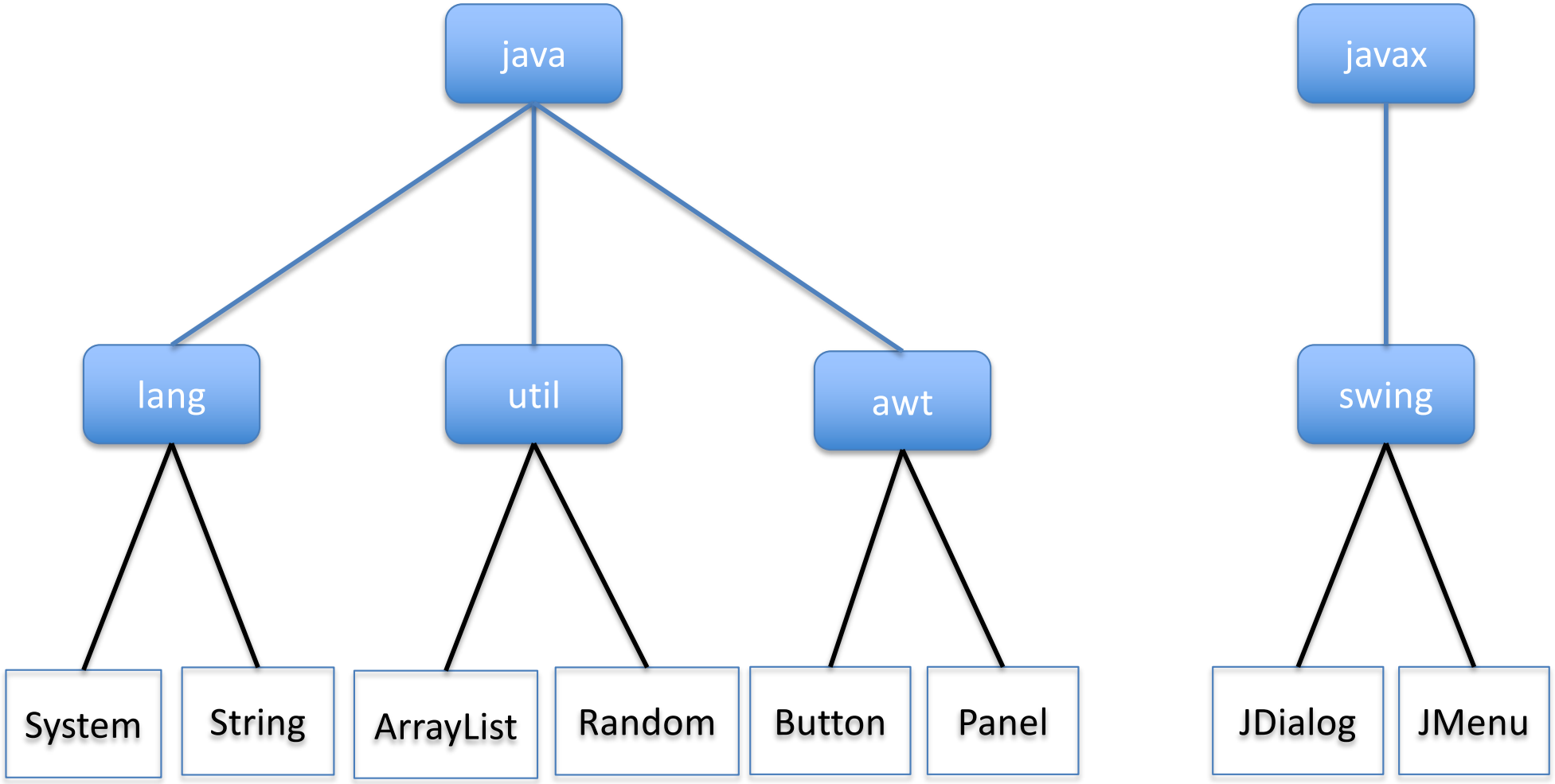
- On Windows machines use:

```
javaw <ClassName>
```

- On UNIX/Linux machines use:

```
java <ClassName> &
```


Packages


java
javax
lang
util
awt
swing
System
String
ArrayList
Random
Button
Panel
JDialog
JMenu

Using Packages

- Classes must be imported before use
- Exception is java.lang

```
import java.util.ArrayList;
import javax.swing.*;
```

- * = all classes in package or child package

Creating Packages

MyClass.java

```
package package1;
public class MyClass {
    //Some code
}
```

OtherClass.java

```
package package2;
import package1.MyClass;
public class OtherClass {
    public static void main(String[] args){
        MyClass mc;
        System.out.println("All OK");
    }
}
```


Compiling with Packages

- Packages are directories with the same name
- Put class files in the package directory
- Child packages are sub-directories
- java looks in the current directory
- -classpath or –cp tells java to look elsewhere as well

Redundant Imports

```
1: import java.lang.System;
2: import java.lang.*;
3: import java.util.Random;
4: import java.util.*;
5: public class ImportExample {
6:  public static void main(String[] args) {
7:    Random r = new Random();
8:    System.out.println(r.nextInt(10));
9:  }
10: }
```

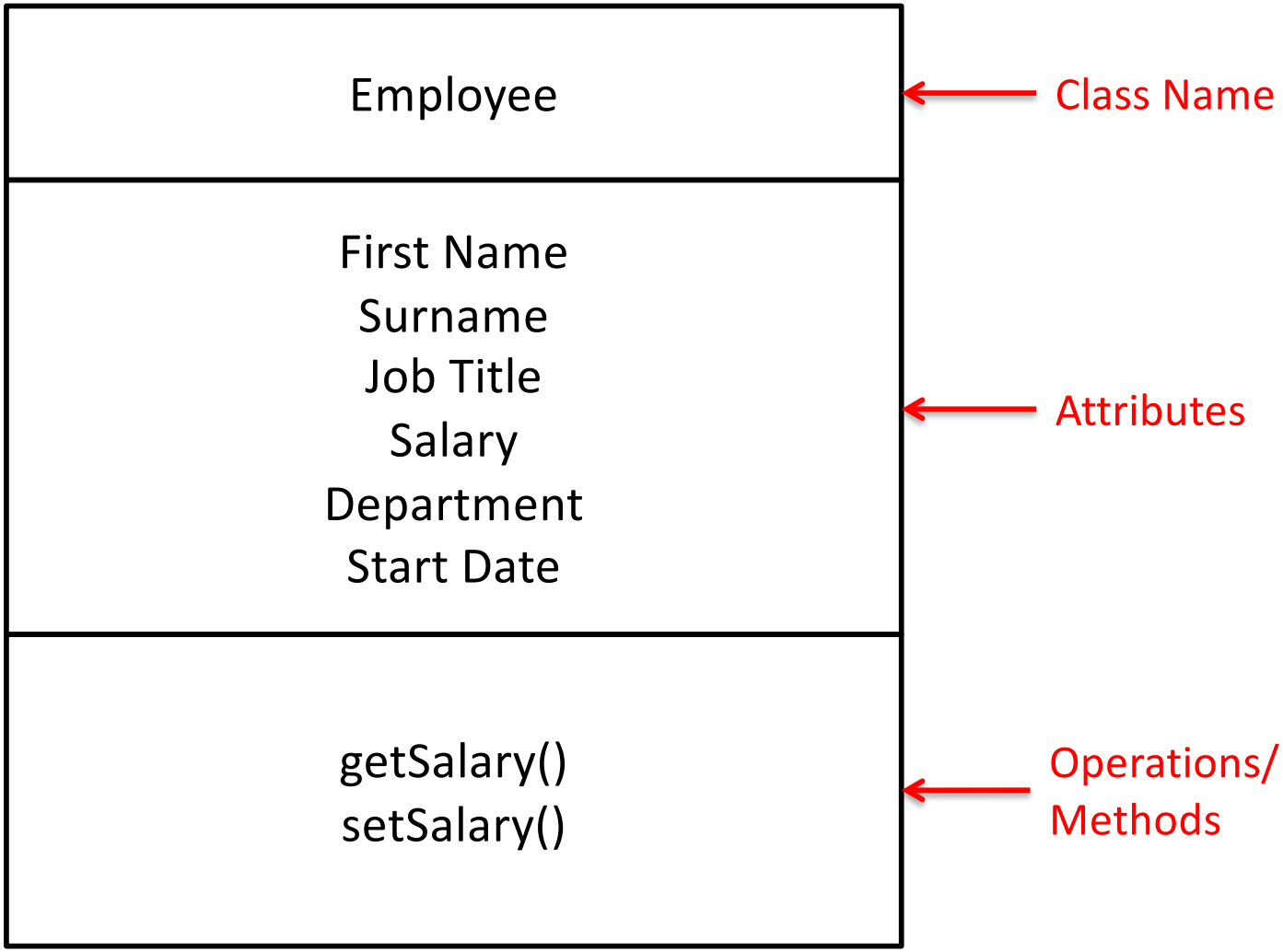

Required Imports

```
public class InputImports {
  public void read(Files files) {
    Paths.get("name");
  }
}
```


Object Example

Alex Jones
Programmer
32500
Development
16/04/2010

Class Example



Classes in Java

```
class Employee {
    String firstName;
    String surname;
    String jobTitle;
    Float salary;
    String department;
    Date startDate;
    String getfirstName(){
        …
    }
    …
}
```


Creating Objects in Java

```
…
Employee emp1;
emp1 = new Employee();
…
…
Employee emp2 = emp1;
…
emp1 = null;
```


Constructors

- Method name matches class name
- No return value (not even void)
- If not present, JVM uses default (do nothing) constructor

```
public class C1 {
   public C1() {
      // Code
   }
}
```


Initializing Variables

```
public class Chicken {
  int numEggs = 0;// initialize on line
  String name;
  public Chicken() {
    name = "Duke";// initialize in constructor
  } }
```


Instance Initializer Blocks

```
public class Chick {
   private String name = "Fluffy";
   { System.out.println("Setting field"); }
   public Chick() {
      name = "Tiny";
      System.out.println("Setting constructor");
   }
   public static void main(String[] args) {
      Chick chick = new Chick();
      System.out.println(chick.name);
   }
}
```


Order of Initialization

```
public class Egg {
  public Egg() {
    number = 5;
  }
  public static void main(String[] args) {
    Egg egg = new Egg();
    System.out.println(egg.number);
  }
  private int number = 3;
  { number = 4; } }
```


Primitive Types

Keyword	Type	Literal Values	Default Value *
`boolean`	true or false	true or false	false
`byte`	8-bit integer	-128 to +127	0
`short`	16-bit integer	-32,768 to +32,767	0
`int`	32-bit integer	-2^{31} to $+2^{31}-1$	0
`long`	64-bit integer	-2^{63} to $+2^{63}-1$	0L
`float`	32-bit floating-point	3.14159F	0.0F
`double`	64-bit floating-point	3.14159265358979	0.0
`char`	16-bit Unicode	‘a’ = 0x61	‘\u0000’ (NUL)

* Local variables must be explicitly initialised and do not default

Declaring and Initialising Variables

```
String name;
name = "Toothpaste";
int count = 10;
MyClass mc = new MyClass();
String s1, s2;
int i1 = 1, i2 = 2;
```


Literals

```
boolean result = true;

char capitalC = 'C';

byte b = 100;

short s = 10000;

int i = 100000;

long l = 10000L;
```


Integer Literals

```
// The number 26, in decimal

int decVal = 26;

//  The number 26, in hexadecimal

int hexVal = 0x1a;

// The number 26, in binary

int binVal = 0b11010;
```


Floating Point Literals

```
double d1 = 123.4;
// same value as d1, but in
scientific notation
double d2 = 1.234e2;
float f1  = 123.4f;
```


Character and String Literals

- May contain any Unicode (UTF-16) characters
- Use Unicode escape sequence such as:
- '\u0108' (capital C with circumflex)
- "S\u00ED Se\u00F1or" (Sí Señor in Spanish)
- 'single quotes' for char literals and "double quotes" for Strings
- Unicode escape sequences may be used elsewhere in a program (such as in field names, for example), not just in char or String literals.

Underscores in Numeric Literals

```
long creditCardNumber = 1234_5678_9012_3456L;
long socialSecurityNumber = 999_99_9999L;
float pi =  3.14_15F;
long hexBytes = 0xFF_EC_DE_5E;
long hexWords = 0xCAFE_BABE;
long maxLong = 0x7fff_ffff_ffff_ffffL;
byte nybbles = 0b0010_0101;
long bytes = 0b11010010_01101001_10010100_10010010;
```


Invalid Underscores

Invalid positions:

- At the beginning or end of a number
- Adjacent to a decimal point in a floating point literal
- Prior to an F or L suffix
- In positions where a string of digits is expected

```
float pi1 = 3_.1415F;
float pi2 = 3._1415F;
long socialSecurityNumber1 = 999_99_9999_L;
int x1 = 5_2;
int x2 = 52_;
int x3 = 5_______2;
int x4 = 0_x52;
int x5 = 0x_52;
int x6 = 0x5_2;
int x7 = 0x52_;
```


Variable Identifiers

- Start with a letter, $ or _
- Thereafter numbers are also allowed
- No reserved words:

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	

Reference Types

- Fields/variables that refer to an object
- Assigning `null` means not currently referring to an object
- N.B. Cannot assign `null` to a primitive type
- Primitive types start with lower case
- java classes start with upper case (recommended for all classes)
- Reference variables are used to access fields and methods of an object

Literals

```
int i = 50;      // Base 10
int i = 070;     // Octal
int i = 0x3F;    // Hexadecimal
int i = 0b10;    // Binary
char c = '\u0065'    // Unicode

int million = 1_000_000;

float f = 123.456F;  // F required
double d = 123.456;

String s = "Hello";
```


Variable Scope

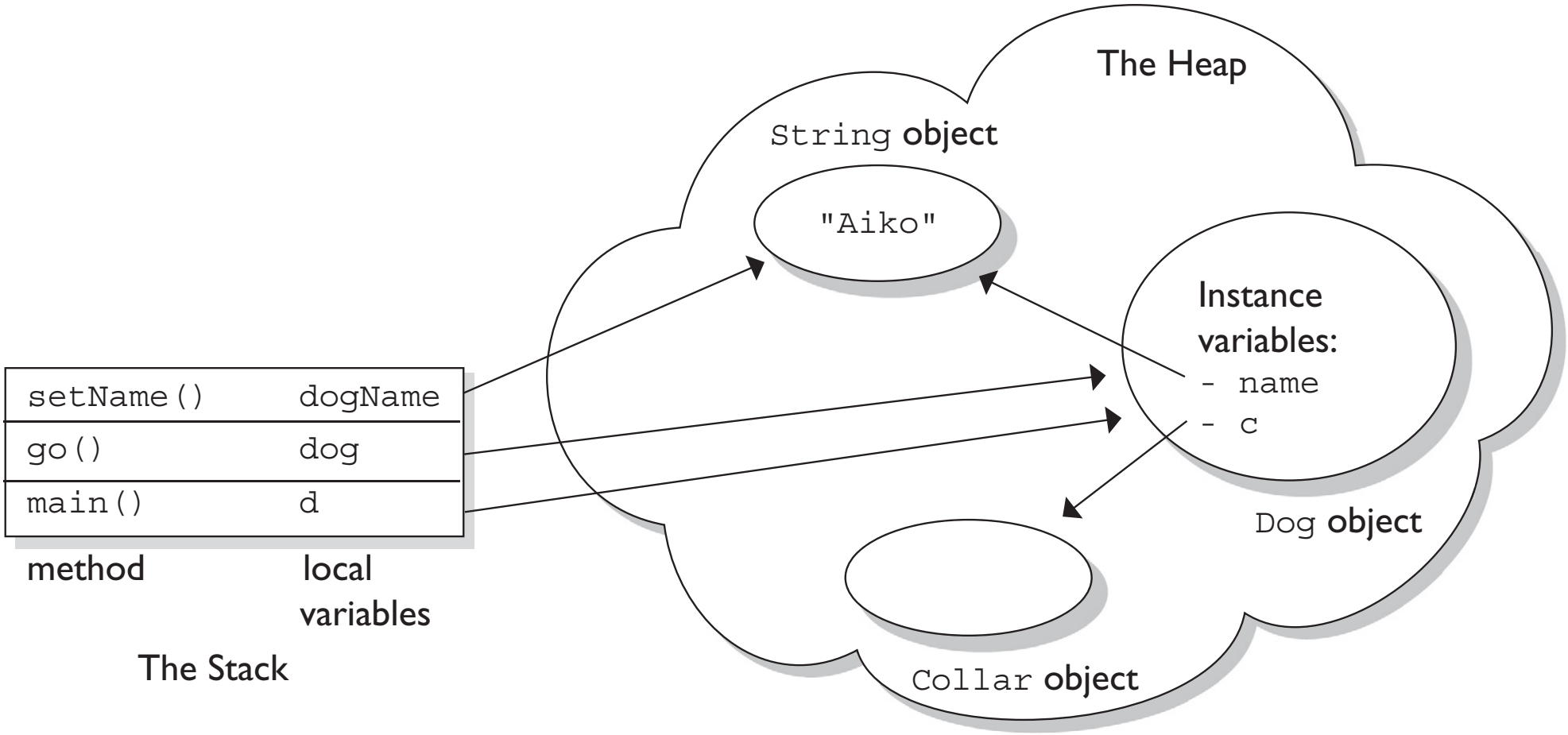
- **Local** – inside a method (including parameters)
- **Instance** – inside a class
- **Class** – inside a class but with `static` keyword meaning it is shared between instances
- Instance and Class variables are automatically initialised
- Code blocks further limit the scope:

```
{ int blockOnly; }
```


Stack and Heap Code

```
 1. class Collar { }
 2.
 3. class Dog {
 4.   Collar c;                          // instance variable
 5.   String name;                       // instance variable
 6.
 7.   public static void main(String [] args) {
 8.
 9.     Dog d;                           // local variable: d
10.     d = new Dog();
11.     d.go(d);
12.   }
13.   void go(Dog dog) {                 // local variable: dog
14.     c = new Collar();
15.     dog.setName("Aiko");
16.   }
17.   void setName(String dogName) {     // local var: dogName
18.     name = dogName;
19.     // do more stuff
20.   }
21. }
```


Stack and Heap Illustration



Stack and Heap Explanation

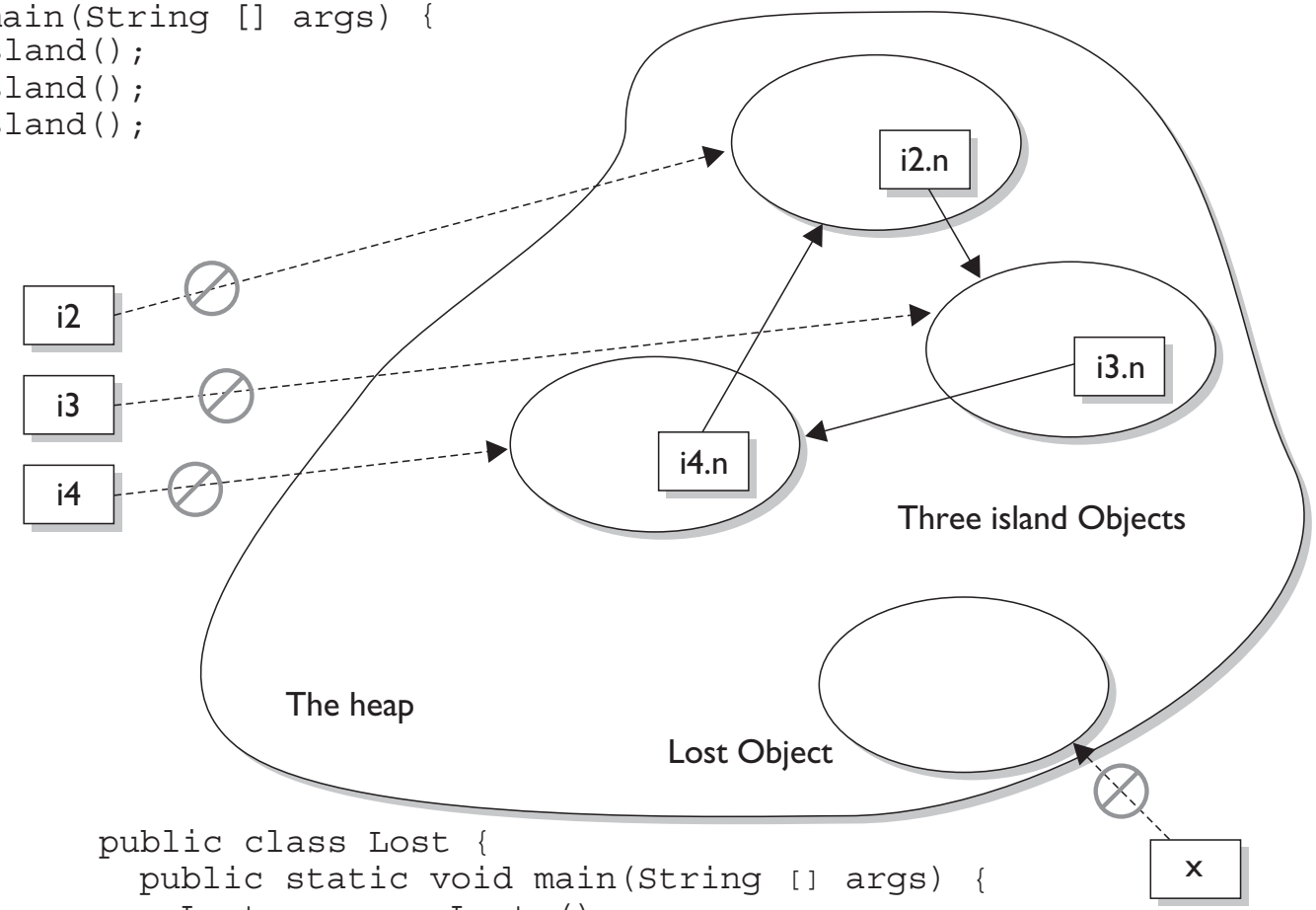
- Line 7—main() is placed on the stack.
- Line 9—Reference variable d is created on the stack, but there's no Dog object yet.
- Line 10—A new Dog object is created and is assigned to the d reference variable.
- Line 11—A copy of the reference variable d is passed to the go() method.
- Line 13—The go() method is placed on the stack, with the dog parameter as
- a local variable.
- Line 14—A new Collar object is created on the heap and assigned to Dog's instance variable.
- Line 17—setName() is added to the stack, with the dogName parameter as its local variable.
- Line 18—The name instance variable now also refers to the String object.
- Notice that two *different* local variables refer to the same Dog object.
- Notice that one local variable and one instance variable both refer to the sameString Aiko.
- After Line 19 completes, setName() completes and is removed from the stack. At this point the local variable dogName disappears, too, although the String object it referred to is still on the heap.

Garbage Collection

- Automatically runs and destroys objects that have no in-scope references pointing to them
- Objects can have a method called finalize() that runs once before they are destroyed

Garbage Collection Example

```
public class Island {
  Island n;
  public static void main(String [] args) {
    Island i2 = new Island();
    Island i3 = new Island();
    Island i4 = new Island();
    i2.n = i3;
    i3.n = i4;
    i4.n = i2;
    i2 = null;
    i3 = null;
    i4 = null;
    doComplexStuff();
  }
}
```



Indicated an active reference

Indicates a deleted reference

```
public class Lost {
  public static void main(String [] args) {
    Lost x = new Lost ();
    x = null;
    doComplexStuff();
  }
}
```


finalize()

- Might not get called
- Definitely won't be called twice

```
public class Finalizer {
    private static List objects = new ArrayList();
    protected void finalize() {
        objects.add(this); // Don't do this
    }
}
```


Ordering Elements in 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

Benefits of Java

- Object Oriented
  - Code is encapsulated in classes
  - Pre-Java languages were procedural
  - Java allows for functional programming within a class/object oriented structure
- Encapsulation
  - Access modifiers protect data from unintended access and modification
- Platform Independent
  - Compiles to bytecode
  - Compiled once for all platforms
  - “write once, run everywhere”
- Robust
  - Prevents memory leaks
  - Manages memory and does garbage collection automatically
  - Bad memory management in C++ is a big source of errors in programs
- Simple
  - Simpler than C++ with no pointers and no operator overloading
- Secure
  - Code runs inside the JVM
  - Creates a sandbox that makes it hard for Java code to do bad things

Java SE8 OCA

Session 2:
Operators and Statements

Java Operators

Operator	Symbols and Usage
Post-unary operators	expression++, expression--
Pre-unary operators	++expression, --expression
Other unary operators	+, -, !
Multiplication, Division and Modulus	*, /, %
Addition and Subtraction	+, -
Bit shift	<<, >>, >>>
Relational	<, >, <=, >=, instanceof
Equality	==, !=
Logical	&, ^, |
Combination logical	&&, ||
Ternary	boolean expression ? expression 1 : expression 2
Assignment	=, +=, -=, *=, /=, %=, &=, ^=, !=, <<=, >>=, >>>=

Primitive Numeric Promotion

- Operands promoted to larger data type
- integer types promoted to floating point
- `byte`, `short` and `char` always promoted to `int` for binary operations
- Result has promoted type

```
int x = 1;
long y = 5;
r = x * y;
```

```
short x = 8;
short y = 4;
short z = x * y;
short z = (short)(x * y)
```


Unary Operators

```
int i = 0;
int x = ++i;
int y = x--;
int z = ++y * 5 - y++;

int j = -1;
int k = -j;
boolean b = !(j == k);
```


Logical Operators

x & y	true	false
true	true	false
false	false	false

x | y	true	false
true	true	true
false	true	false

x ^ y	true	false
true	false	true
false	true	false

Relational and Equality Operators

```
int i = 0, j = 1;
double d = 1.0, e = 2.0;
MyClass mc = new MyClass(), mc2 = mc;

boolean b1 = (i > j);
boolean b2 = (d < i);
boolean b3 = (mc instanceof MyClass);

boolean b4 = (d == j);
boolean b5 = (e == 1);
boolean b6 = (mc2 == mc);
```


Assignment Operators

```
int x = 1.0; // DOES NOT COMPILE
short y = 1921222; // DOES NOT COMPILE
int z = 9f; // DOES NOT COMPILE
long t = 192301398193810323; // DOES NOT COMPILE

int x = (int)1.0;
short y = (short)1921222; // Stored as 20678
int z = (int)9l;
long t = 192301398193810323L;
```


Casting Primitives

```
int x = (int)1.0;
short y = (short)1921222; // Stored as 20678
int z = (int)9l;
long t = 192301398193810323L;
```

Overflow:

```
System.out.print(2147483647+1); // -2147483648
```


if-then-else Statement

```
if(boolExp) {
      // code executed if boolExp is true
} else if(boolExp2) {
      // code executed if boolExp2 is true
      // AND boolExp is false
} else {
      // code executed if none of the
      // preceding boolean expressions
      // evaluates to true
}
```


Ternary Operator

```
int y = 5;
int z;
if(y > 2) {
   z = 2 * y;
} else {
   z = 5 * y;
}

int y = 5;
int z = (y > 2) ? (2 * y) : (5 * y);
```

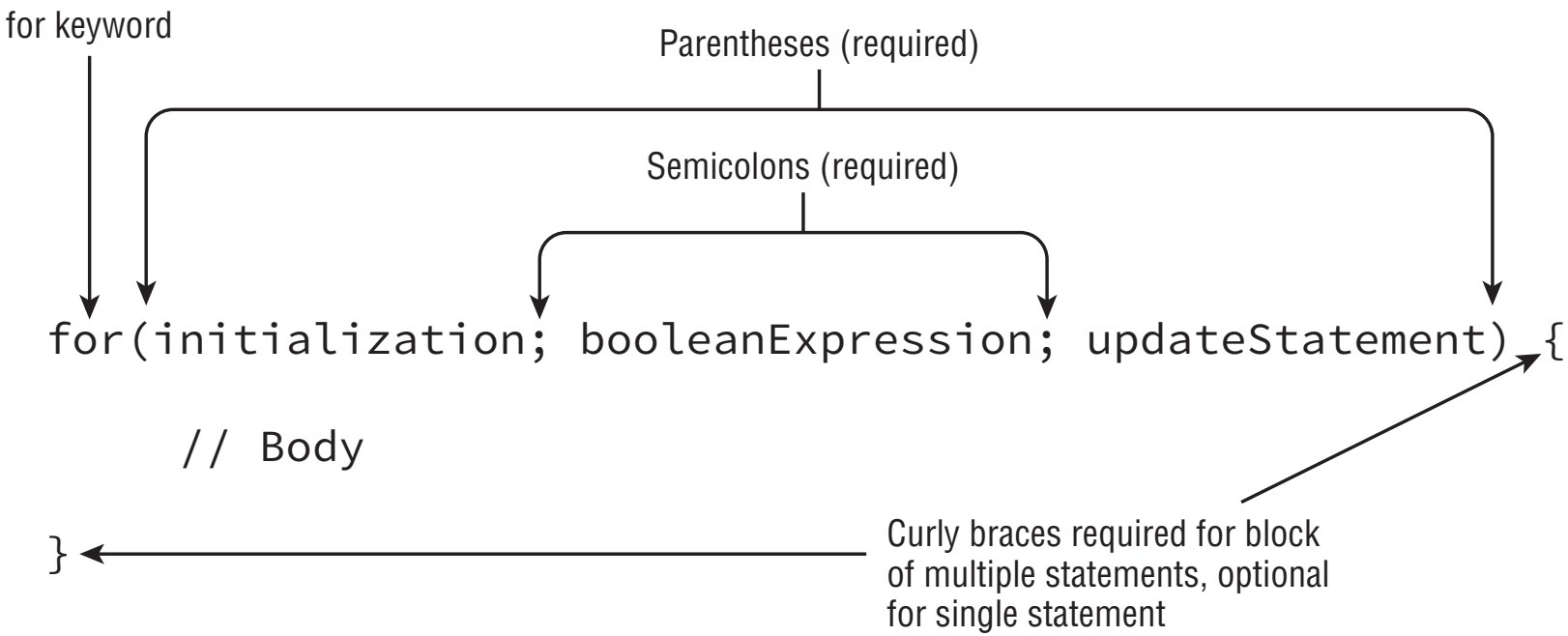

`while` Statement

```
while(booleanExpression) {
     // code
}

do {
     // code
} while(booleanExpression);
```


for Statement

```
for(init; booleanExp; update) {
     // code
}

for(dataType variableName : collection) {
     // code
}

break optLabel;
continue optLabel;
```


for Structure



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

for Examples

```
for( ; ; ) {
  System.out.println("Hello World");
}
```

```
int x = 0;
for(long y = 0, z = 4; x < 5 && y < 10; x++, y++) {
  System.out.print(y + " ");
}
System.out.print(x);
```

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

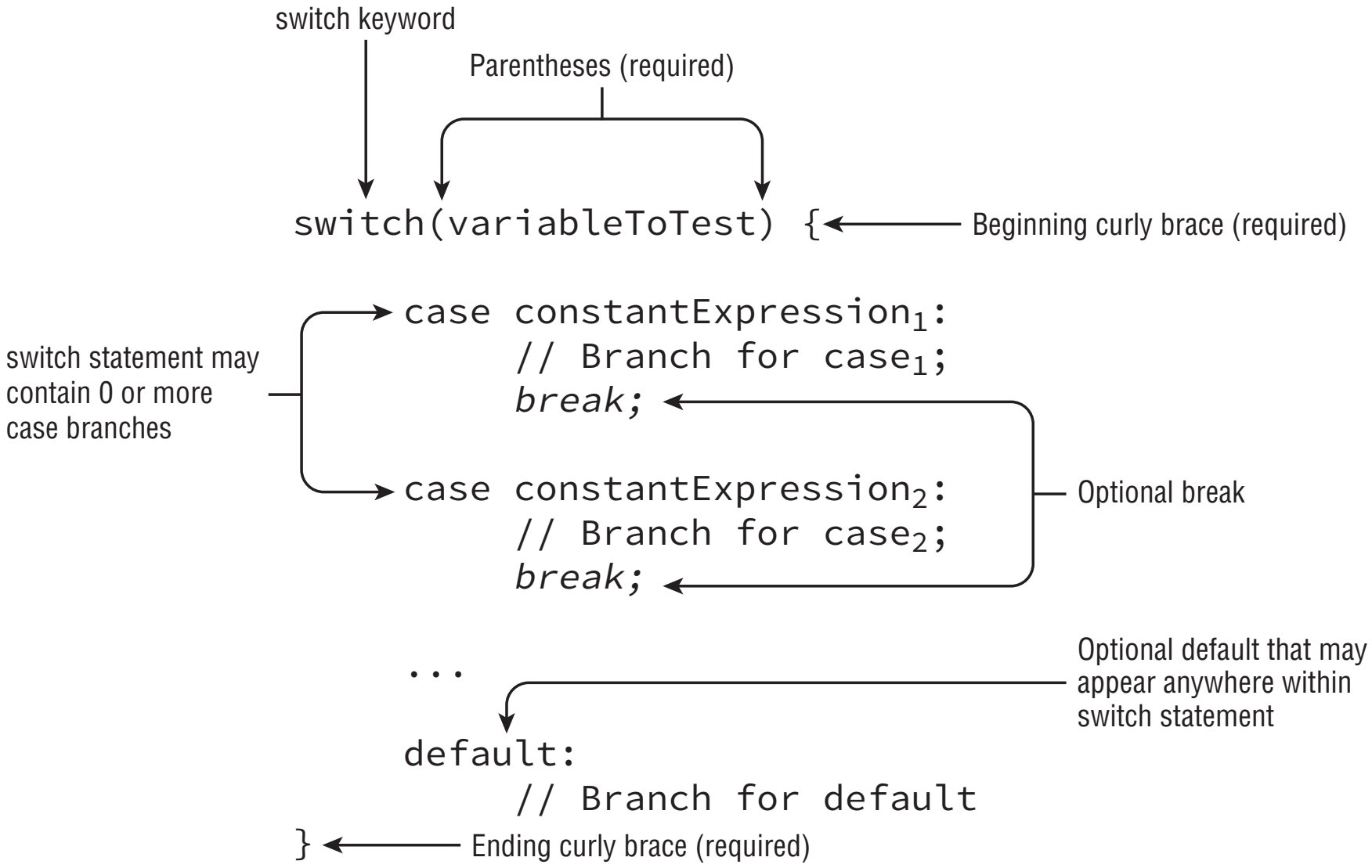
```
for(long y = 0, int x = 4; x < 5 && y<10; x++, y++) {   // DOES NOT COMPILE
  System.out.print(x + " ");
}
```

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


switch Statement

```
switch(expression){
   case constantExpression1:
      // code
      break;
   case constantExpression2:
      // code
      break;
   ...
   default:
      //code
}
```


Switch Structure



break Statement (1)

```
int dayOfWeek = 5;
switch(dayOfWeek) {
   default:
      System.out.println("Weekday");
      break;
   case 0:
      System.out.println("Sunday");
      break;
   case 6:
      System.out.println("Saturday");
      break;
}
```


break Statement (2)

```
int dayOfWeek = 5;
switch(dayOfWeek) {
    case 0:
       System.out.println("Sunday");
    default:
       System.out.println("Weekday");
    case 6:
       System.out.println("Saturday");
    break;
}
```


Legitimate Case Values

```
private int getSortOrder(String firstName, final String lastName) {
    String middleName = "Patricia"; final String suffix = "JR";
    int id = 0;
    switch(firstName) {
        case "Test":
            return 52;
        case middleName: // DOES NOT COMPILE
            id = 5; break;
        case suffix:
            id = 0; break;
        case lastName: // DOES NOT COMPILE
            id = 8; break;
        case 5: // DOES NOT COMPILE
            id = 7; break;
        case 'J': // DOES NOT COMPILE
            id = 10; break;
        case java.time.DayOfWeek.SUNDAY: // DOES NOT COMPILE
            id=15; break;
    }
    return id;
}
```


Data Types Supported

- int and Integer
- byte and Byte
- short and Short
- char and Character
- String
- enum values

Break and Continue Summary

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

Java SE8 OCA

Session 3:
Core Java APIs

Concatenation

- For numbers, `+` means addition
- For `String`s, `+` means concatenation and anything else is converted to a `String`
- As usual expressions are evaluated left to right

String Indexing

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
M	y		S	t	r	i	n	g		E	x	a	m	p	l	e

```
String s = "My String Example";

s.length()
s.charAt(index)
s.indexOf(char/String)
s.indexOf(char/String, index)
s.substring(beginIndex)
s.substring(beginIndex, endIndex)
```


Other `String` Methods

```
String s = "My String Example";

s.toLowerCase()
s.toUpperCase()
s.equals("My String Example")
s.equalsIgnoreCase("my string example)
s.startsWith("M")
s.endswith("le")
s.contains("Exam")
s.replace(' ', '_')
s.replace("My", "Your");
"  String of pearls   ".trim()
s.toLowerCase().indexOf('s')
```


StringBuilder Class

```
StringBuilder sb = new StringBuilder();
String firstName = “Zuri”
String lastName = “Martino”
sb.append(firstName);
sb.append(‘ ’);
sb.append(lastName);
```


StringBuilder Constructors

```
StringBuilder sb1 = new StringBuilder();
StringBuilder sb2 = new StringBuilder("animal");
StringBuilder sb3 = new StringBuilder(10);
```

StringBuilder changes its own state and returns a reference to itself:

```
4: StringBuilder a = new StringBuilder("abc");
5: StringBuilder b = a.append("de");
6: b = b.append("f").append("g");
7: System.out.println("a=" + a);
8: System.out.println("b=" + b);
```


`StringBuilder` Methods

- Same as `String`: `charAt()`, `indexOf()`, `length()` and `substring()`

```
.append(anyDataType)
.insert(offset, String)

.toString() // Inherited from Object Class
```

Only way to compare StringBuilders:

```
sb1.toString().equals(sb2.toString())
```


Examples

```
StringBuilder sb = new
StringBuilder().append(1).append('c');
sb.append("-").append(true);
System.out.println(sb); // 1c-true


3: StringBuilder sb = new StringBuilder("animals");
4: sb.insert(7, "-"); // sb = animals-
5: sb.insert(0, "-"); // sb = -animals-
6: sb.insert(4, "-"); // sb = -ani-mals-
7: System.out.println(sb);
```


Other StringBuilder Methods

delete() and deleteCharAt()

```
StringBuilder delete(int start, int end)
StringBuilder deleteCharAt(int index)
```

Examples:

```
StringBuilder sb = new StringBuilder("abcdef");
sb.delete(1, 3); // sb = adef
sb.deleteCharAt(5); // throws an exception
```

reverse()

```
StringBuilder reverse()
```


Equality

```
StringBuilder one = new StringBuilder();
StringBuilder two = new StringBuilder();
StringBuilder three = one.append("a");
System.out.println(one == two); // false
System.out.println(one == three); // true

String x = "Hello World";
String y = "Hello World";
String z = new String("Hello World”);
System.out.println(x == y); // true (String pool)
System.out.println(x == z); // false (New object created)

String x = "Hello World";
String z = " Hello World".trim();
System.out.println(x == z); // false (computed at runtime)
```


Arrays

- Simple ordered list allowing duplicates:

```
int[] intArray = new int[10];
```

- Uninitialised arrays contain the default value (`0` for `int` etc.)
- Index starts at `0` like `String` objects

```
int[] intArray = new int[] {3, 7, 19};
int intArray []; // Also allowed
int... intArray; // Known as varargs
```


Array Contents

- Primitive arrays contain primitives
- Reference arrays contain references to the object type declared
- Find number of slots with `.length`
- `java.util.Arrays` object has useful methods:

```
Arrays.sort(myArray)
Arrays.binarySearch(myArray, searchItem)
Arrays.toString(myArray)
```


Searching Arrays

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

```
3: int[] numbers = {2,4,6,8};
4: System.out.println(Arrays.binarySearch(numbers, 2)); // 0
5: System.out.println(Arrays.binarySearch(numbers, 4)); // 1
6: System.out.println(Arrays.binarySearch(numbers, 1)); // -1
7: System.out.println(Arrays.binarySearch(numbers, 3)); // -2
8: System.out.println(Arrays.binarySearch(numbers, 9)); // -5
```


Multidimentional Arrays

```
int[][] intArray2D;
int intArray2D[][];
int[] intArray2D[];
int[] intArray2D[], intArray3D[][];
```

- Asymmetric multidimentional arrays:

```
int[][] intAsymArray = {{1, 2}, {3},
                        {4, 5, 6}};
```


`ArrayList` Class

- Ordered sequence with duplicates allowed, like arrays
- Arrays have a fixed number of elements
- `java.util.ArrayList` can change size as needed

```
ArrayList al1 = new ArrayList();
ArrayList al2 = new ArrayList(5);
ArrayList al3 = new ArrayList(al2);
```


Generics

```
ArrayList<String> al5 = new ArrayList<String>();
ArrayList<String> al5 = new ArrayList<>();
```

- `ArrayList` implements a `List` interface:

```
List<String> al6 = new ArrayList<>();
```


ArrayList Methods: add

```
ArrayList al = new ArrayList();
Boolean add(E element)
void add(int index, E element)
   // E = Object or contents of <>

ArrayList<String> al = new ArrayList<>();
al.add("Smith"); // index omitted
a1.add(0, "Jones");  // adds at index 0
```


ArrayList Methods: remove

```
ArrayList<String> al = new ArrayList<>();
boolean remove(Object object)
E remove(int index)

al.add("Smith");
a1.add("Jones");
System.out.println(al.remove("Jones")); // true
System.out.println(al.remove("Jones")); // false
System.out.println(al.remove(0));       // Smith
```


ArrayList Methods: set

```
ArrayList<String> al = new ArrayList<>();
E set(int index, E newElement)

al.add("Smith");
a1.add("Jones");

System.out.println(al.set(0, "Singh")); // Smith

System.out.println(al.size);            // 2

System.out.println(al.remove(0));       // Singh
System.out.println(al.remove(0));       // Jones
System.out.println(al.isEmpty());       // true
```


ArrayList Other Methods

```
ArrayList<String> al = new ArrayList<>();


void clear()
boolean contains(Object object)
boolean equals(Object object)
```


Converting between array and `List`

```
ArrayList<String> al = new ArrayList<>();
    // ...code to initialise al

String[] strArray = al.toArray(new String[0]);
/* a new array of the same runtime type is allocated to
 * store the elements of the list
*/

String[] strArray = {“s1”, “s2”};
List<String> list = Arrays.asList(strArray);
       // Creates a backed fixed-sized List

Collections.sort(myArrayList);  // Sorts an ArrayList
```


Wrapper Classes

Primitive	Wrapper class	Example Constuctor
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(1L)
float	Float	new Float(1.0F)
double	Double	new Double(1.0)
char	Character	new Character('x')

```
Boolean.parseBoolean("true");
Boolean.valueOf("true");
```

- Autoboxing converts primitives to wrappers and vice-versa

Converting from a 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

Autoboxing

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


3: List<Integer> heights = new ArrayList<>();
4: heights.add(null);
5: int h = heights.get(0);         // NullPointerException


List<Integer> numbers = new ArrayList<>();
numbers.add(1);
numbers.add(2);
numbers.remove(1);                 // removes index 1
System.out.println(numbers);       // outputs 1

numbers.remove(new Integer(1))     // removes Integer object
                                   // containing 1
```


Dates and Times

```
import java.time.*;
LocalDate ld = LocalDate.of(2015, Month.JULY, 1);
LocalDate ld = LocalDate.of(2015, 7, 1);
LocalTime lt = LocalTime.of(9, 0)
               // Seconds and nanoseconds also available
LocalDateTime = LocalDateTime.of(ld, lt);
               // Can also use (y, m, d, h, m, s, ns)

.now()        // Returns current date, time or both
.plusDays(d)  // Also plusYears etc
.minusDays(d) // These return objects but don't change
               // the original value
               // Also quite common to see these chained
```


Period Class

```
Period p = Period.ofMonths(3);
       // Also .ofYears, Weeks and Days
Period p2 = Period.of(1, 2, 3); // y, m, d
       // Use Duration for <= 1 day
LocalDate ld = LocalDate.of(2015, 1, 1);
System.out.println(ld.plus(p));

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
```


Period Between Dates

```
LocalDate today = LocalDate.now ( ) ;
    LocalDate birthday = LocalDate.of (1970, 1, 1) ;

    Period p = Period.between ( birthday, today ) ;

    System.out.println ( "You are "
        + p.getYears ( )
        + " years, "
        + p.getMonths ( )
        + " months, and "
        + p.getDays ( )
        + " days old!" ) ;
```


Examples

Period.ofXXX methods are static methods:

```
Period wrong = Period.ofYears(1).ofWeeks(1); // every week
```

Really like writing the following:

```
Period wrong = Period.ofYears(1);
wrong = Period.ofWeeks(7);

7: System.out.println(date.plus(period)); // 2015-02-20
8: System.out.println(dateTime.plus(period));
                                        // 2015-02-20T06:15
9: System.out.println(time.plus(period));
                              // UnsupportedTemporalTypeException
```


Getting Date Time Information

```
LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
System.out.println(date.getDayOfWeek()); // MONDAY
System.out.println(date.getMonth());     // JANUARY
System.out.println(date.getYear());      // 2020
System.out.println(date.getDayOfYear()); // 20
```


Formatting Dates and Times

```
import java.time.*
import java.time.format.DateTimeFormatter
LocalDateTime ldt = LocalDateTime.of(2015, 1, 1, 9, 30);

ldt.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME)
```

DateTimeFormatter static methods:

```
.ofLocalizedDate(FormatStyle.SHORT)
.ofLocalizedTime(FormatStyle.SHORT)
.ofLocalizedDateTime(FormatStyle.SHORT)
.ofLocalizedDateTime(FormatStyle.MEDIUM)

.ofPattern("MMMM dd, yyyy, hh:mm")
```


Examples

```
LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
LocalTime time = LocalTime.of(11, 12, 34);
LocalDateTime dateTime = LocalDateTime.of(date, time);

DateTimeFormatter shortF = DateTimeFormatter
.ofLocalizedDateTime(FormatStyle.SHORT);

DateTimeFormatter mediumF = DateTimeFormatter
.ofLocalizedDateTime(FormatStyle.MEDIUM);

System.out.println(shortF.format(dateTime));
                                    // 1/20/20 11:12 AM
System.out.println(mediumF.format(dateTime));
                                    // Jan 20, 2020 11:12:34 AM
```

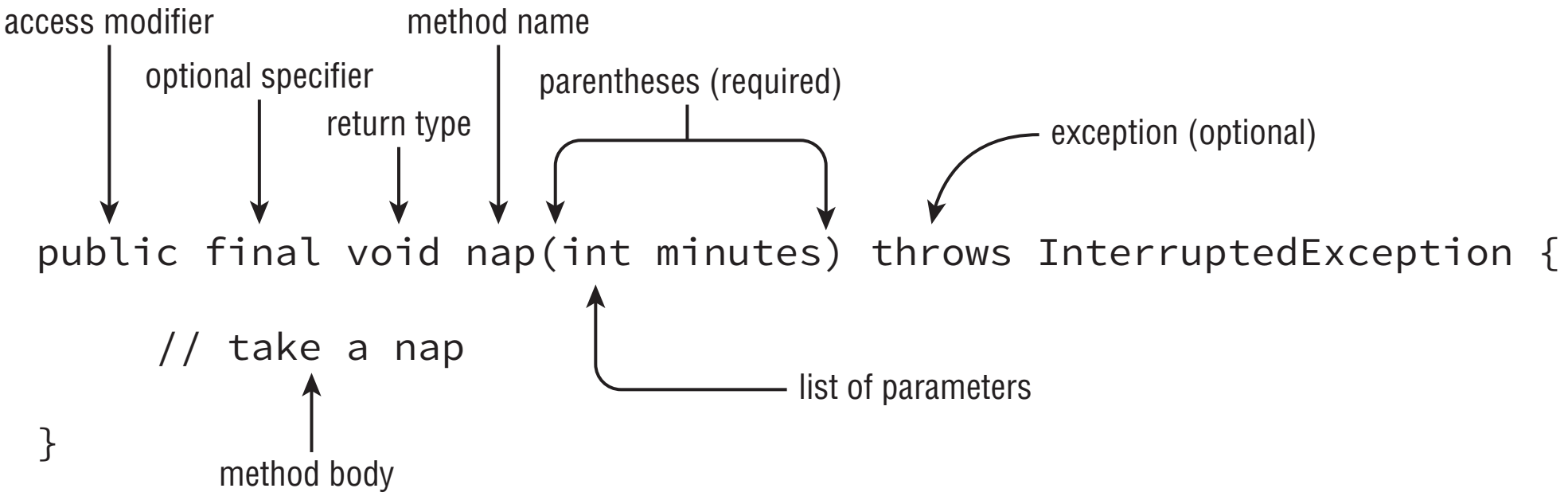

Parsing Dates and Times

```
import java.time.*
import java.time.format.*

DateTimeFormatter f ...
LocalDateTime dt ...

f.format(dt);
dt.format(f);
LocalDate.parse("01 01 2015", f); //Uses f
LocalTime.parse("09:30");         // Uses default
```


Java SE8 OCA

Session 4:
Methods and Encapsulation

Method Declaration



Method Signature – Access Modifiers

```
public final void m(int i) throws xE
```

Access Modifiers:

- `public`
- `private`
- `protected`
- Default (no access modifier)

Method Signature – Optional Specifiers

```
public final void m(int i) throws xE
```

Optional Specifiers:

- `static`
- `abstract`
- `final`
- `synchronized`
- `native`
- `strictfp`

Method Signature – Return Type & Name

```
public final void m(int i) throws xE
```

Return Type: Must return this type or if void must not return anything

Method names have same restrictions as variable names

Method Signature – Param List and Exception

```
public final void m(int i) throws xE
```

Parameter List:

- Parentheses must be present.
- Any parameters must have data types
- Separated by commas
- Comply with naming rules
- Become local variables

Exceptions:

- Indicate if something goes wrong
- Multiple allowed separated by commas

Method Signature – Varargs

```
public final void m(int... i)
```

Varargs:

- Only one per method
- Must be last in list

Method call:

- Pass in list
- Pass in array
- Local variable is an array

Access Modifiers: `private`

package p1

```
package p1;
public class C1{
    private void m1(){
        // Code
    }
}
```

```
package p1;
public class C2{
    // Code
}
```

package p2

```
package p2;
public class C3{
    private void method(){
        / Code
    }
}
```

```
package p2;
import p1.C1;
public class C4 extends C1
{
    // Code
}
```


Access Modifiers: `private`

package p1

```
package p1;
public class C1{
    private void m1(){
        // Code
    }
}
```

```
package p1;
public class C2{
    // Code
}
```

package p2

```
package p2;
public class C3{
    private void method(){
        / Code
    }
}
```

```
package p2;
import p1.C1;
public class C4 extends C1
{
    // Code
}
```


Access Modifiers: Default

package p1

```
package p1;
public class C1{
    void m1(){
        // Code
    }
}
```

```
package p1;
public class C2{
    // Code
}
```

package p2

```
package p2;
public class C3{
    private void method(){
        / Code
    }
}
```

```
package p2;
import p1.C1;
public class C4 extends C1
{
    // Code
}
```


Access Modifiers: Default

package p1

```
package p1;
public class C1{
    void m1(){
        // Code
    }
}
```

```
package p1;
public class C2{
    // Code
}
```

package p2

```
package p2;
public class C3{
    private void method(){
        / Code
    }
}
```

```
package p2;
import p1.C1;
public class C4 extends C1
{
    // Code
}
```


Access Modifiers: protected

package p1

```
package p1;
public class C1{
    protected void m1(){
        // Code
    }
}
```

```
package p1;
public class C2{
    // Code
}
```

package p2

```
package p2;
public class C3{
    private void method(){
        / Code
    }
}
```

```
package p2;
import p1.C1;
public class C4 extends C1
{
    // Code
}
```


Access Modifiers: `protected`

package p1

```
package p1;
public class C1{
    protected void m1(){
        // Code
    }
}
```

```
package p1;
public class C2{
    // Code
}
```

package p2

```
package p2;
public class C3{
    private void method(){
        / Code
    }
}
```

```
package p2;
import p1.C1;
public class C4 extends C1
{
    // Code
}
```


Access Modifiers: `public`

package p1

```
package p1;
public class C1{
    public void m1(){
        // Code
    }
}
```

```
package p1;
public class C2{
    // Code
}
```

package p2

```
package p2;
public class C3{
    private void method(){
        / Code
    }
}
```

```
package p2;
import p1.C1;
public class C4 extends c1
{
    // Code
}
```


Access Modifiers: `public`

package p1

```
package p1;
public class C1{
    public void m1(){
        // Code
    }
}
```

```
package p1;
public class C2{
    // Code
}
```

package p2

```
package p2;
public class C3{
    private void method(){
        / Code
    }
}
```

```
package p2;
import p1.C1;
public class C4 extends c1
{
    // Code
}
```


`static` Methods and Fields

```
public class C1{
   String s;
   static int count;
   public static final int MAX = 10;
   void m1(int n) {
      int x;
      for(int i = 0; i < n; i++){
         System.out.println(i);
      }
   }
   static String toString() {
      // Code
   }
}
```


Using `static` Methods and Fields

```
import C1;
System.out.println(C1.count);
C1.m1(5);
C1 c = new C1();
System.out.println(c.count);
c.m1(5);
```

- Static members cannot call instance members
- Instance members can call statics

Accessing Static Variable and Methods

Put the class name before the method or variable:

```
System.out.println(Koala.count);
Koala.main(new String[0]);
```

Can use an instance of the object to call a static method:

```
5: Koala k = new Koala();
6: System.out.println(k.count); // k is a Koala
7: k = null;
8: System.out.println(k.count); // k is still a Koala
```


Static vs. Instance

```
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();
      second();
      third(); // DOES NOT COMPILE
   }
}
```


Static Initialization

Run when the class is first used:

```
private static final int NUM_SECONDS_PER_HOUR;
static {
    int numSecondsPerMinute = 60;
    int numMinutesPerHour = 60;
    NUM_SECONDS_PER_HOUR = numSecondsPerMinute *
    numMinutesPerHour;
}
```


static imports

```
import static java.lang.Math.*
double h = sqrt((o * o) + (a * a));
double c = 2.0 * PI * r
```

- Careful not to import two members with the same name

Method Overloading

```
public void m1(int i){ }
public int m1(int i, String s) { }
String m1(int i, String s, int x) { }
```

- Parameters must be different or JVM will not know which to use

```
long m1(int i, String s) { } // No good
```


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 void fly(int numMiles) { }
public int fly(int numMiles) { } // DOES NOT COMPILE


public void fly(int numMiles) { }
public static void fly(int numMiles) { }
                                        // DOES NOT COMPILE
```


Using Varargs

```
public int m1(String[] s) { }
public int m1(String... s) { }
```

- These have identical signatures
- Call either version with an array:

```
String[] strArray = {“a”, “b”, “2”};
m1(strArray);
```

- Varargs version requires separate parameters:

```
m1(“a”, “b”, “c”);
```

```
public void fly(int[] lengths) { }
public void fly(int... lengths) { } // DOES NOT COMPILE
```


Autoboxing

```
public void m1(int i) { }
public void m1(Integer i) { }

public void m2(String s) { }
public void m2(Object o) { }

public void m3(int i) { }
public void m3(long l) { }
```


Choosing the Right Overloaded Method

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) {}`

Constructors

- Method name matches class name
- No return value (not even void)
- If not present, JVM uses default (do nothing) constructor

```
public class C1 {
   public C1() {
      // Code
   }
}
```


Constructors with Arguments

- Instantiation provides arguments:

```
C1 c = new C1("London", 1);
```

- Constructor receives local method arguments:

```
public class C1 {
   String name;
   int number;
   public C1(String name, int number) {
      this.name = name;       //Initialisation
      this.number = number;   //Initialisation
   }
}
```


Overloading Constructors

- Same rules as overloading any method:

```
public class C1 {
   ...
   public C1(String name, int number) { ... }
   public C1(int number) { ... }
   }
```

- If any constructor coded, there is no default “no args” constructor provided by the compiler

Constructor Chaining

- Avoids code duplication:

```
public class C1 {
    int number;
    String name, type;
    public C1(String s) { this(s, 0, “U”); }
    public C1(String s, int n) { this(s, n, “U”); }
    public C1(String s, int n, String t) {
        this.name = s;
        this.number = n;
        this.type = t;
    }
}
```


Constructor Initialisation

- Constructors can initialise `final` fields:

```
import static java.lang.Math.*;
public class Circle {
    private final int AREA;
    public Circle (int radius) {
        AREA = PI * radius * radius;
    }
}
```

- Order of initialisation:
  1. Superclass
  2. `static` declarations and `static` initialisers in order
  3. Instance declarations and instance initialisers in order
  4. Constructor

Final Fields

```
public class MouseHouse {
    private final int volume;
    private final String name = "The Mouse House";
    public MouseHouse(int length, int width, int height) {
        volume = length * width * height;
    }
}
```

- Constructor is part of the initialization process
- It is allowed to assign final instance variables
- By the time the constructor completes, all final instance variables must have been set

Order of Initialization

```
public class InitializationOrderSimple {
    private String name = "Torchie";
    { System.out.println(name); }
    private static int COUNT = 0;
    static { System.out.println(COUNT); }
    static { COUNT += 10; System.out.println(COUNT); }
    public InitializationOrderSimple() {
        System.out.println("constructor");
    }
}
```

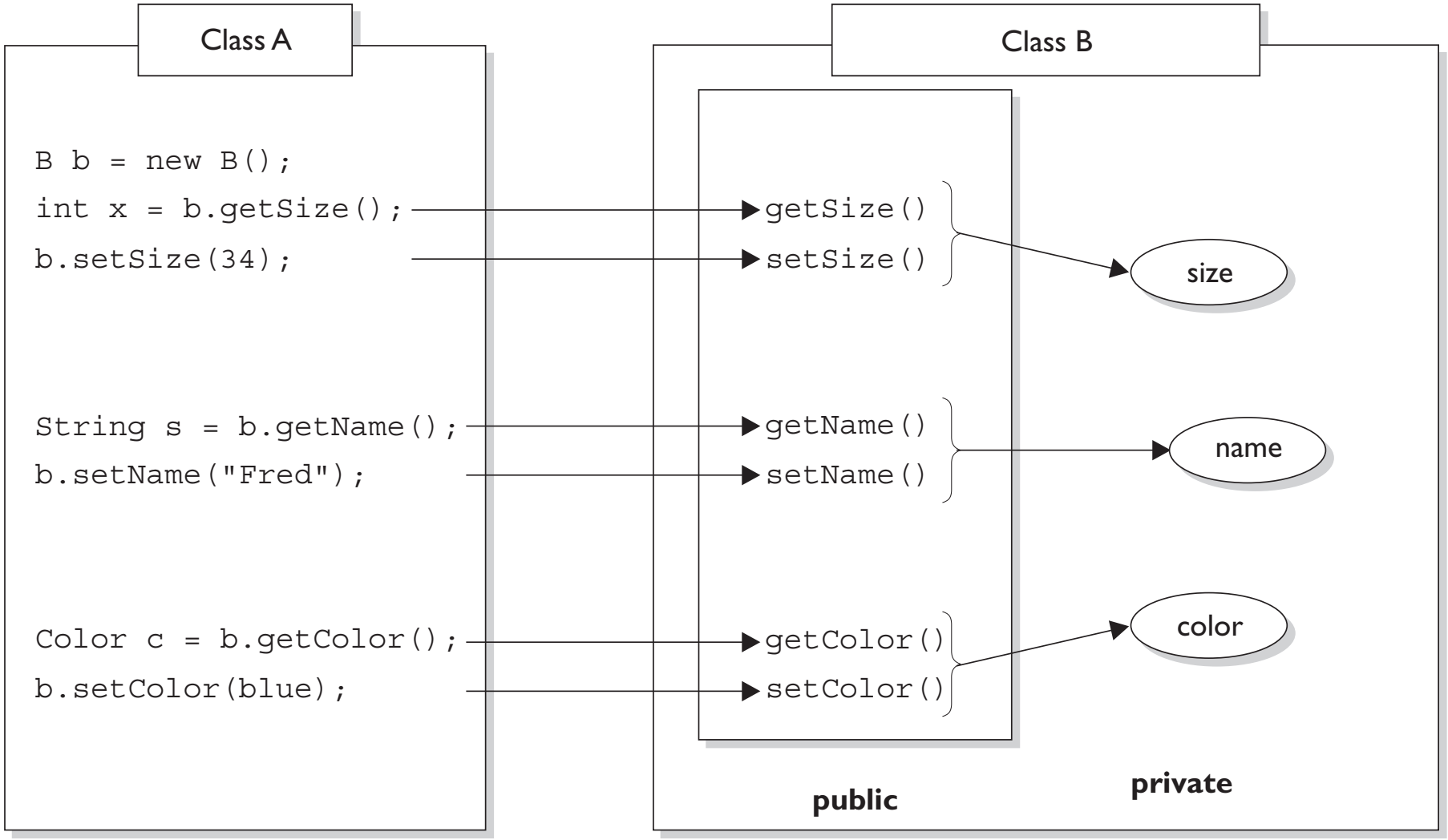
1. Initialize superclass
2. Static variable declarations and static initializers in order
3. Instance variable declarations and instance initializers in order
4. The constructor.

Encapsulation

- Hides the implementation of a class
- Allows control of fields and methods:

```
public class C1 {
   private int number;
   private String name;
   public int getNumber() {
      return number;
   }
   public void setNumber(int number) {
      if(number > 0)
         this.number = number
   }
}
```


Encapsulation



Encapsulation?

```
public class circle{
    float radius;
    float area;
    public float getRadius(){
        return radius;
    }
    public float getArea() {
        return area;
    }
    public void setRadius(float radius){
        this.radius = radius;
        this.area = Math.PI * r * r;
    }
}
```


Encapsulation Naming Conventions

- Follow the JavaBeans rules:
  - Properties (fields) are private
  - Properties start with a lower case letter
  - Getter method begins with is for boolean properties
  - Getter method begins with get for all other properties
  - Setter method begins with set
  - Follow is/get/set with the property name starting with an upper case letter

Immutable Classes

- No changes allowed:

```
public class C1 {
    private int number;
    private String name;
    public int getNumber() {
        return number;
    }
    public C1(int number) {
        this.number = number;
    }
}
```


Return Types in Immutable Classes

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

public Mutable(StringBuilder b) {
    builder = new StringBuilder(b);
}
public StringBuilder getBuilder() {
    return new StringBuilder(builder);
}
```


Lambda Expressions

- Expressions that can be passed to a method
- Contain code that will be executed later
- Here is an example class:

```
public class Employee {
   private String name;
   private String department;
   private boolean isPermanent;
   ...  // Fields are initialised
   ...   // Getter and setter methods provided
   ...   // toString method provided
}
```


Functional Interface

- Interface with a single method
- Here is an example interface:

```
public interface FilterEmployee {
   boolean test(Employee e);
}
```

- Here is a class that implements it:

```
public class CheckPermanent implements
                        FilterEmployee {
   public boolean test(Employee e) {
      return e.isPermanent();
   }
}
```


Method that Accepts an Interface

- Here is a method which accepts a parameter of type `FilterEmployee`:

```
private static void print(List<Employee>
                employees, FilterEmployee filter) {
    for (Employee e : employees) {
        if (filter.test(e))
            System.out.println(e);
    }
}
```


Using the `print` method

- Here is an example of code that uses this method:

```
public class EmployeeSearch {
    public static void main(String[] args) {
    List<Employee> employees = new ArrayList<Employee>();
    employees.add(new Employee("Jones", "Sales", true);
    employees.add(new Employee("Smith", "Sales", false);
    employees.add(new Employee("Singh", "Finance", true);
    print(employees, new CheckPermanent());
    }
}
```

- This passes the class `CheckPermanent` to the `print` method

Using a Lambda in Place of a Class

- Here is an alternative using a Lambda expression:

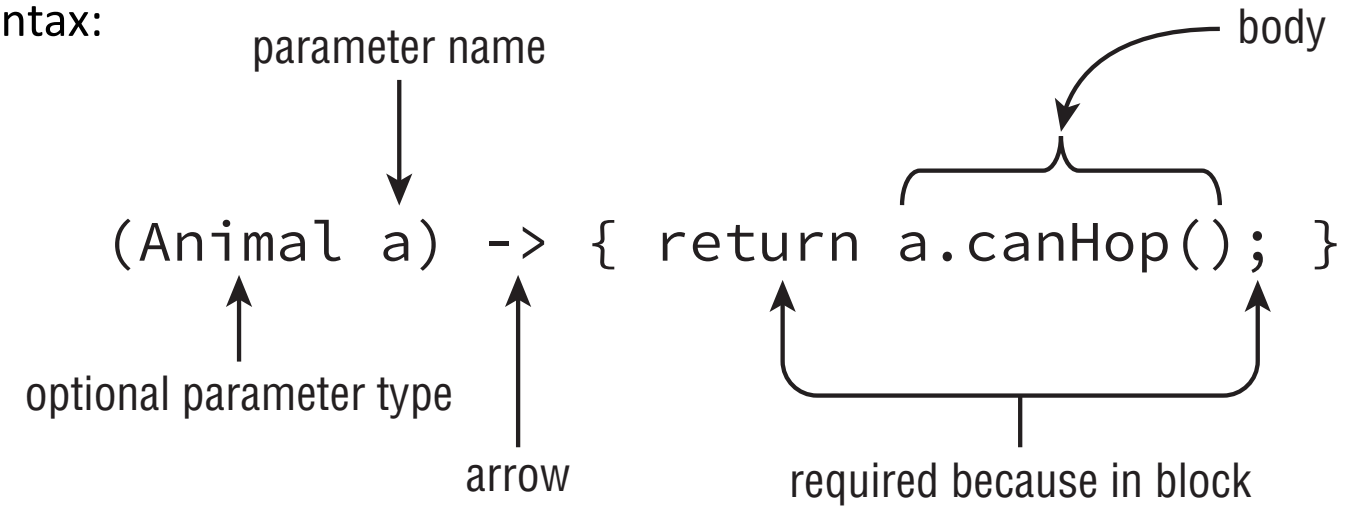
```
print(employees, e -> e.isPermanent());
```

- This passes some code instead of a class
- Changing the code changes the test:

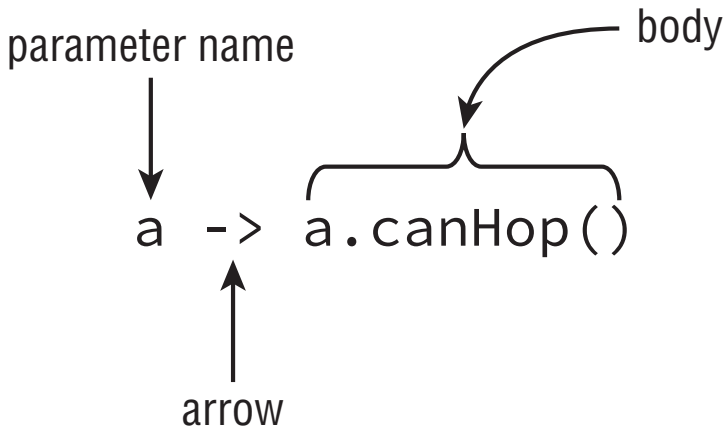
```
print(employees, e ->
                e.getDepartment().equals("Sales")
```


Lambda Syntax

Maximum Syntax:



Minimum Syntax:



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


print(a, b -> a.startsWith("test")); // DOES NOT COMPILE
print(a -> { a.startsWith("test"); }); // DOES NOT COMPILE
print(a -> { return a.startsWith("test") });
                                              // DOES NOT COMPILE


(a, b) -> { int a = 0; return 5;} // DOES NOT COMPILE

(a, b) -> { int c = 0; return 5;} // OK
```


Lambda Expressions, Predicates

- Java provides an `interface` called `Predicate`

```
public interface Predicate<T> {
   boolean test(T t);
}
```

- This can be used instead of writing your own:

Animal Entity Class

```
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; }
}
```


Trait Testing Interface

```
public interface CheckTrait {
    boolean test(Animal a);
}

public class CheckIfHopper implements CheckTrait {
    public boolean test(Animal a) {
        return a.canHop();
    }
}
```


Trait Testing Interface

```
public class TraditionalSearch {
    public static void main(String[] args) {
    List<Animal> animals = new ArrayList<Animal>();
                                        // list of animals
    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, new CheckIfHopper());
                            // pass class that checks trait
}
private static void print(List<Animal> animals,
                                        CheckTrait checker) {
    for (Animal animal : animals) {
    if (checker.test(animal))
        System.out.print(animal + " ");
    }
    System.out.println(); }
}
```

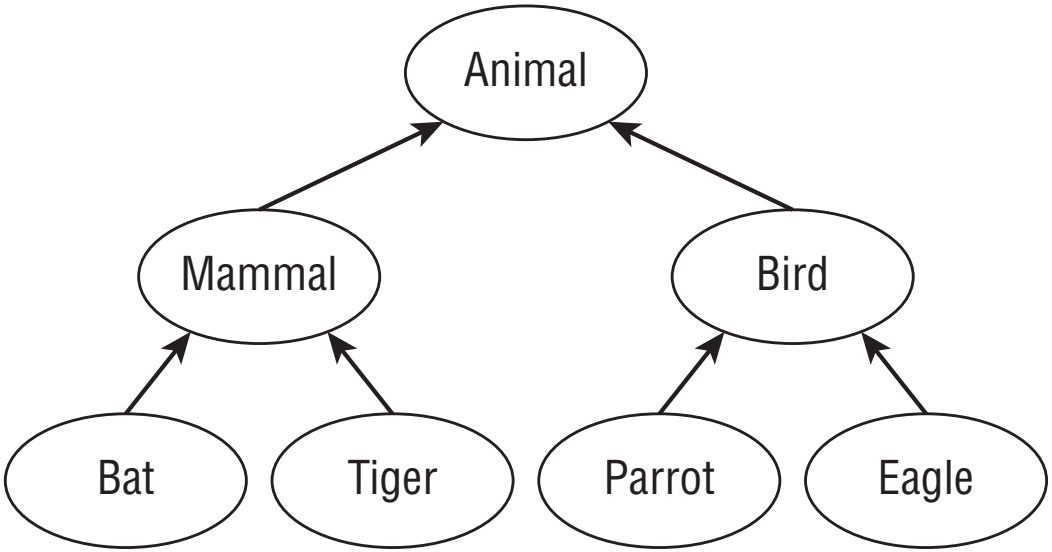

Predicate Version

```
import java.util.*;
import java.util.function.*;
public class PredicateSearch {
   public static void main(String[] args) {
   List<Animal> animals = new ArrayList<Animal>();
   animals.add(new Animal("fish", false, true));
   print(animals, a -> a.canHop());
}
private static void print(List<Animal> animals,
                          Predicate<Animal>↵ checker) {
   for (Animal animal : animals) {
      if (checker.test(animal))
         System.out.print(animal + " ");
      }
      System.out.println();
   }
}
```

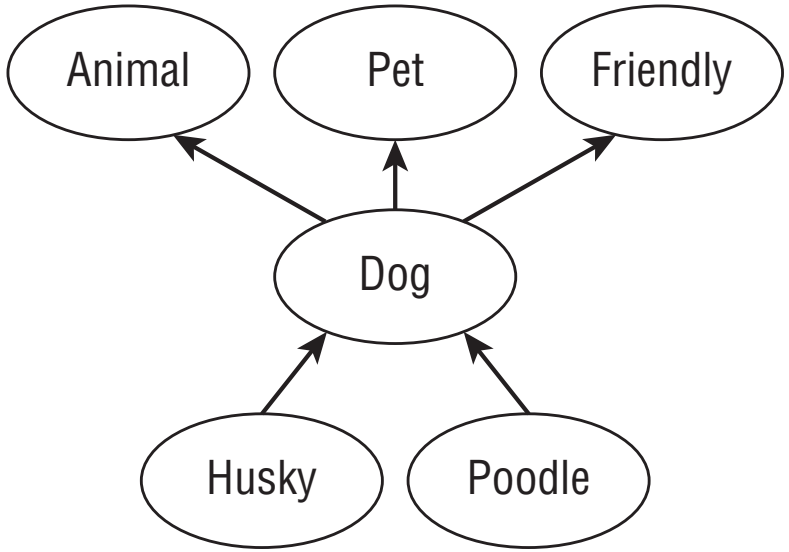

Java SE8 OCA

Session 5:
Class Design

Types of Inheritance

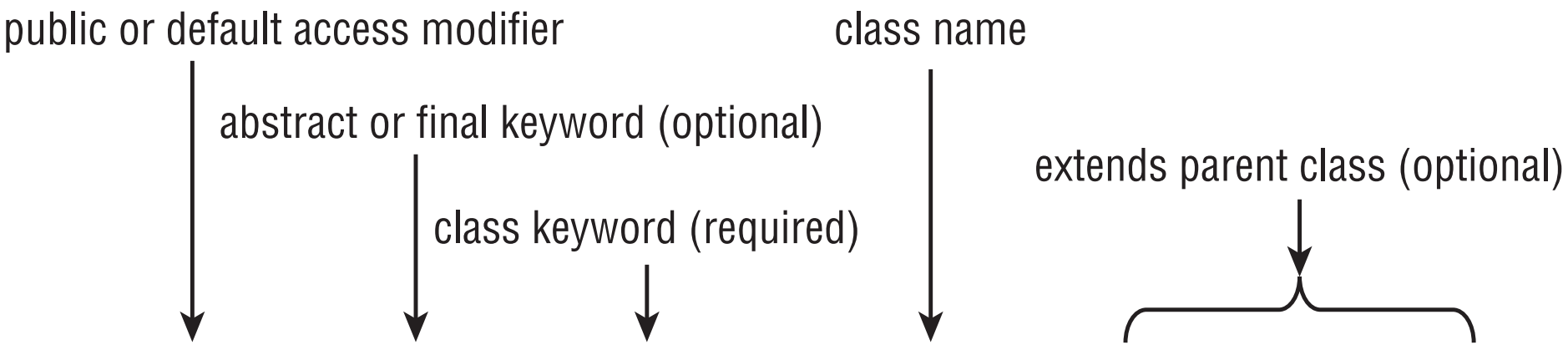


Single Inheritance



Multiple Inheritance

Inheriting from a Super Class



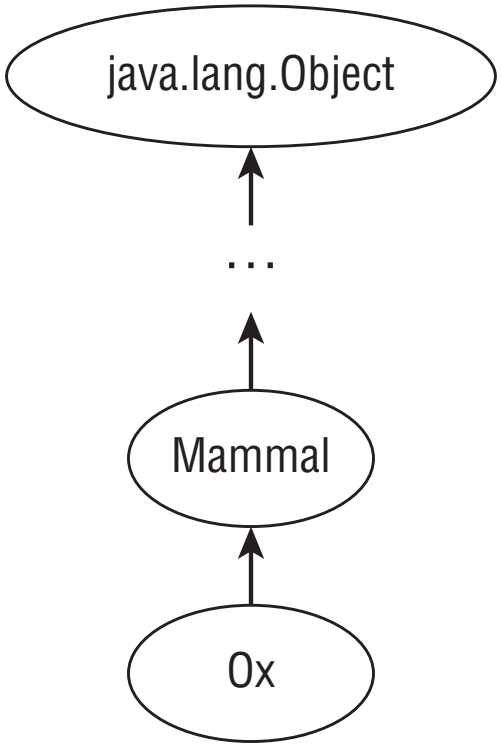

```
public abstract class ElephantSeal extends Seal {

    // Methods and Variables defined here

}
```


Inheritance Example

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

public class Lion extends Animal {
  private void roar() {
    System.out.println("The "+getAge()+" year old lion says: Roar!");
  }
}
```

```
System.out.println("The "+age+" year old lion says: Roar!");
// DOES NOT COMPILE
```


All Classes Inherit from Object



Class Inheritance

```
public class c2 extends c1 {
    // Code
}
```

- All classes inherit directly from `Object` without `extends` keyword
- Compiler adds `extends java.lang.Object`
- Ultimately all classes inherit from the `Object` class

Class Inheritance: Parent/Superclass

```
package hr;
public class Employee {
    private String name, department;
    private boolean permanent;
    public Employee(String name, String department, boolean permanent)
{
        this.name = name;
        this.department = department;
        this.permanent = permanent;
    }
    public String getName() {  return name; }
    public String getDepartment() { return department; }
    public boolean isPermanent() { return permanent; }
    public String toString() {
        return name + ", " + department + ", " +
                             (permanent ? "permanent" : "non-permanent");
    }
}
```


Class Inheritance: Child/Subclass

```
package hr;
public class Intern extends Employee {
   public Intern(String name, String department,
                                 boolean permanent) {
      super(name, department, permanent);
   }
   private String supervisor;
   public String getSupervisor() {
      return supervisor;
   }
   public void setSupervisor(String supervisor) {
      this.supervisor = supervisor;
   }
}
```


Child/Subclass Contents

- Subclasses contain all members of the superclass
- However `private` variables cannot be accessed directly (use getter/setter methods)

`class/interface` Access Modifiers

```
public class C1 { ... }  // Use c1 anywhere

class C2 { ... }  // Use c2 in package
                  // Also in subclasses
```

- Only one `public class` or `interface` per file

Constructors

- Every `class` has at least one constructor
- If none coded then compiler adds a no-args constructor
- Subclass constructors call a superclass constructor with their first statement
- If none specified the compiler adds a call to an accessible no-args constructor
- If there is no accessible no-args constructor the code will not compile
- Calls to `this` or `super` must be the first line in any constructor

Compiler Enhancements

```
public class Donkey {
}
```

```
public class Donkey {
  public Donkey() {
  }
}
```

```
public class Donkey {
  public Donkey() {
    super();
    }
}
```


Coding Constructors

```
public class Mammal {
  public Mammal(int age) {
  }
}

public class Elephant extends Mammal {  // DOES NOT COMPILE
}
```

```
public class Mammal {
  public Mammal(int age) {
  }
}

public class Elephant extends Mammal {
  public Elephant() {  // DOES NOT COMPILE
  }
}
```


Constructor Calling

```
public class GrandParent {
    public GrandParent() { System.out.println("GrandParent - No-args");}
}
class Parent extends GrandParent {
    public Parent() { System.out.println("Parent - No-args"); }
}
class Child extends Parent {
    public Child() {
            System.out.println("Child - No-args");
    }
}
class Tester {
    public static void main(String[] args) {
        GrandParent gp = new GrandParent();
        Parent p = new Parent();
        Child c = new Child();
    }
}
```


Constructor Definition Rules

1. 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().
2. The super() call may not be used after the first statement of the constructor.
3. If no super() call is declared in a constructor, Java will insert a no-argument super() as the first statement of the constructor.
4. 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.
5. If the parent doesn't have a no-argument constructor, the compiler requires an explicit call to a parent constructor in each child constructor.

Inherited Class Members

```
class Parent {
    protected int value;
    private String name;
    public Parent(String name) { this.name = name; }
    public String getName() { return name; }
}
public class Child extends Parent {
    private int childValue = 2;
    public Child(String name) { super(name); this.value = 1; }
    public String toString() {
        return getName() + ", " + value + ", " + childValue;
        // Could use this.value or super.value
        // this.childValue is OK but not super.childValue
    }
}
```

- super() calls the superclass constructor
- super. refers to a member of the superclass

Inheriting Methods

```
public class GrandParent {
    public void m1(){ System.out.println("GrandParent m1"); }
}
class Parent extends GrandParent {
    public void m1(){ System.out.println("Parent m1"); }
}
class Child extends Parent {
    public void m1(String msg){ System.out.println("Child m1: " + msg); }
}
class Tester {
    public static void main(String[] args) {
        GrandParent gp = new GrandParent();
        Parent p = new Parent();
        Child c = new Child();
        gp.m1();
        p.m1();
        c.m1();
        c.m1("Hello");
    }
}
```


Overriding Methods Rules

- Subclass/Child method must:
  1. have the same signature
  2. be at least as accessible
  3. not throw any new or broader checked exceptions
  4. return the same type or a subclass (“covariant return types”)

- A different signature is used for overloading
- `private` methods cannot be overridden but can be redefined in the subclass
- `static` methods may be “hidden” but this is deprecated

Hiding Variables

```
public class Rodent {
    protected int tailLength = 4;
    public void getRodentDetails() {
        System.out.println("[parentTail="+tailLength+"]");
}}
public class Mouse extends Rodent {
    protected int tailLength = 8;
    public void getMouseDetails() {
        System.out.println("[tail="+tailLength
                  +",parentTail="+super.tailLength+"]"); }
    public static void main(String[] args) {
        Mouse mouse = new Mouse();
        mouse.getRodentDetails();
        mouse.getMouseDetails();
} }
```

```
[parentTail=4] [tail=8,parentTail=4]
```


`abstract` Classes

- Cannot be instantiated
- May contain `abstract` methods that must be overridden (implemented)
- May contain normal methods including `main`
- May be extended by other `abstract` classes

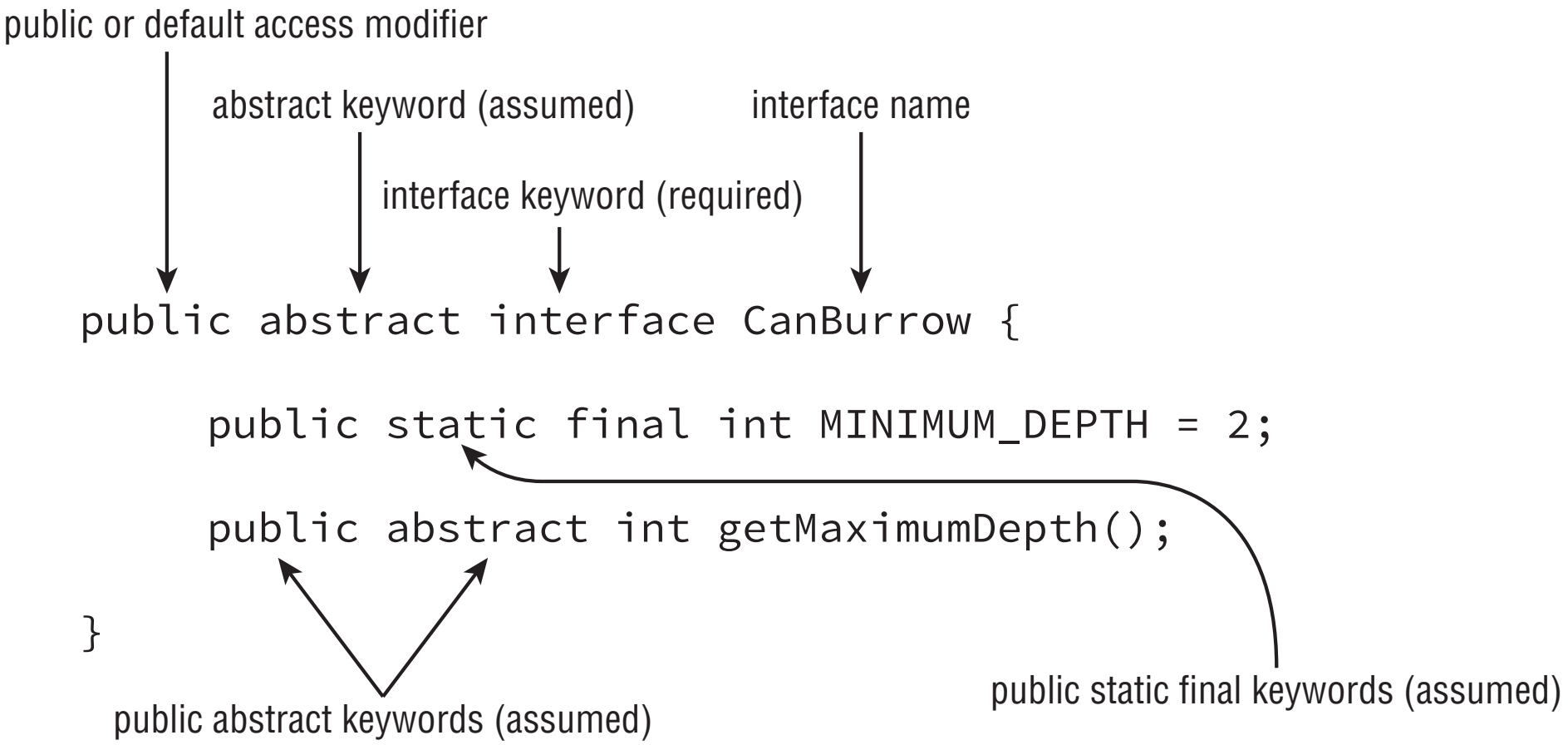
```
public abstract class Asset {
    protected long registerNumber;
    protected int classification;
    protected String description;
    public abstract void assignAsset();
 }
```


Interfaces

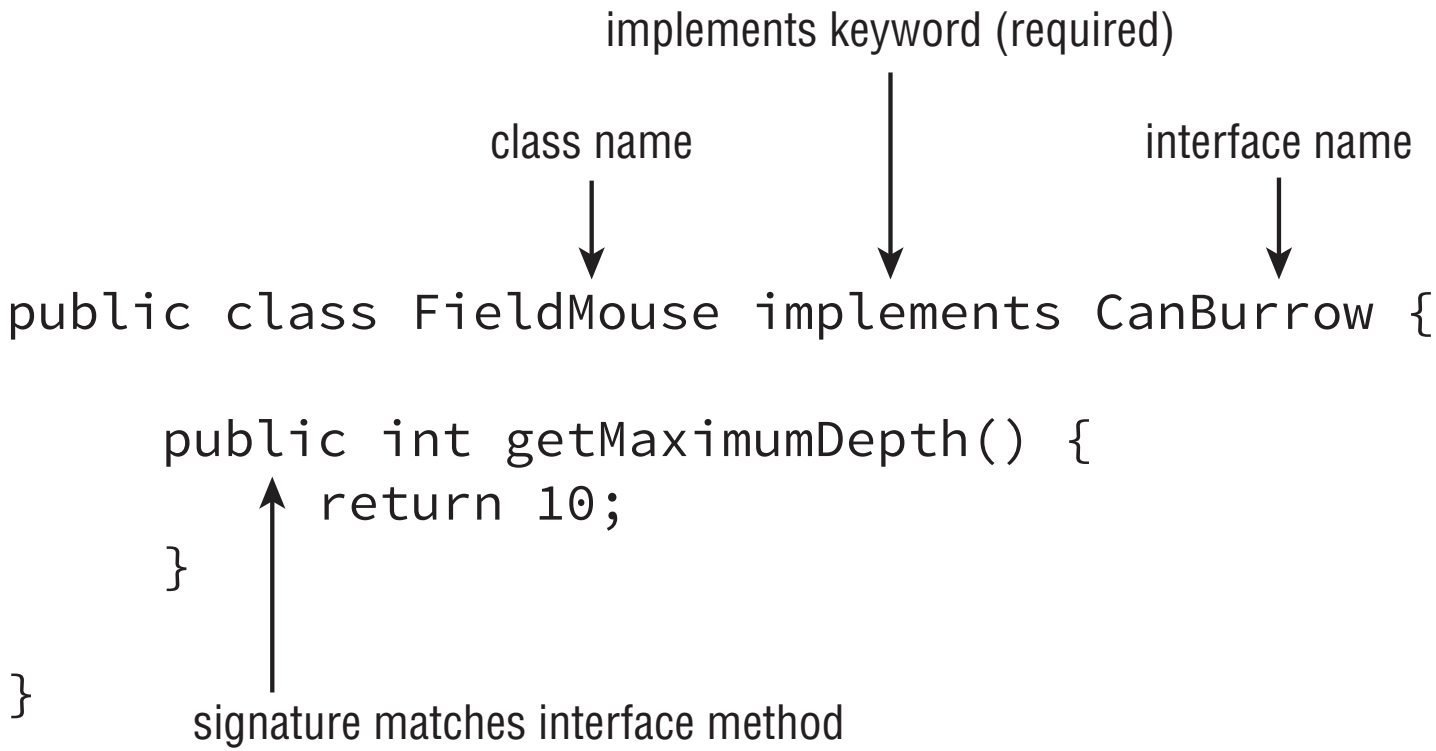
- An `interface` is like an `abstract class`
- Has `abstract` methods and constants
- Classes use `implements` keyword
- A `class` may implement multiple interfaces

```
public abstract interface Assignable {
   public abstract void assign();
   public static final int MAX_ASSIGNMENTS = 100;
}

public class ProjectWork implements Assignable {
   public void assign() { /* Code */ }
}
```


Interface Definition



Implementing Interfaces



Inheriting an Interface

```
public interface HasTail {
    public int getTailLength();
}

public interface HasWhiskers {
    public int getNumberOfWhiskers();
}

public interface Seal extends HasTail, HasWhiskers { }
```


Interface Variables

- Assumed to be public, static, and final.
- Marking as private or protected will trigger a compiler error
- So will marking any variable as abstract
- Value must be set when it is declared since it is marked as final

```
public interface CanSwim {
    int MAXIMUM_DEPTH = 100;
    final static boolean UNDERWATER = true;
    public static final String TYPE = "Submersible";
}
```

Effectively becomes:

```
public interface CanSwim {
    public static final int MAXIMUM_DEPTH = 100;
    public static final boolean UNDERWATER = true;
    public static final String TYPE = "Submersible";
}
```


Inheritance Rules

- A class can implement zero or more interfaces
- Non-abstract and `abstract` classes can extend each other
- An `interface` can extend another `interface`

Abstract Class Rules

- Abstract classes:
  1. may not be instantiated
  2. may have any number of abstract or non-abstract methods
  3. may not be marked private or final
  4. inherit all abstract methods of classes they extend
- The first concrete class that extends an abstract class must implement all its abstract classes including inherited ones

Abstract Method Rules

- Abstract methods:
  1. may only be defined in abstract classes
  2. may not be private or final
  3. must not have a body
  4. must be implemented with the same name, signature and accessibility (just like overriding)

Interface Rules

- Interfaces:
  1. may not be instantiated
  2. have public or default access and are assumed to be abstract and therefore may not be marked as final, private or protected
  3. may have zero to many methods which are assumed to be abstract and public and therefore may not be marked as final, private or protected

Default Interface Methods (Java SE 8)

```
public interface Interface1 {
   int method1(int i);
   public default int method2(){
      return 0;
   }
}
```

- May be overridden either as abstact or default
- Classes inheriting two default methods with same name must override

Static Interface Methods (Java SE 8)

```
public interface Interface1 {
   public static int method1()
      return 0;
   }
}
```

- Not inherited by classes
- Used by referring to interface name (or reference variable of interface type

```
public class Class1 implements Interface1 {
   public void useMethod1() {

System.out.println(Interface1.method1());
   }
}
```


Polymorphism

```
class Test {
  public static void main(String [] args) {
    Test t1 = new Test();
    Test t2 = new Test();
    if (!t1.equals(t2))
      System.out.println("they're not equal");
    if (t1 instanceof Object)
      System.out.println("t1's an Object");
  }
}
```

Produces:

```
they're not equal
t1's an Object
```


Re-use

```
class GameShape {
  public void displayShape() {
    System.out.println("displaying shape");
  }
  // more code
}

class PlayerPiece extends GameShape {
  public void movePiece() {
    System.out.println("moving game piece");
  }
  // more code
}


public class TestShapes {
  public static void main (String[] args) {
    PlayerPiece shape = new PlayerPiece();
    shape.displayShape();
    shape.movePiece();
  }
}
```


Specialised Subclasses

```
class GameShape {
    public void displayShape() {
        System.out.println("displaying shape");
    }
    // more code }
class PlayerPiece extends GameShape {
    public void movePiece() {
        System.out.println("moving game piece");
    }
    // more code }
class TilePiece extends GameShape {
    public void getAdjacent() {
        System.out.println("getting adjacent tiles");
    }
    // more code
}
```


Polymorphism

```
public class TestShapes {
    public static void main (String[] args) {
        PlayerPiece player = new PlayerPiece();
        TilePiece tile = new TilePiece();
        doShapes(player);
        doShapes(tile);
    }
    public static void doShapes(GameShape shape) {
        shape.displayShape();
    }
}
```

outputs:

```
displaying shape
displaying shape
```


Polymorphism

```
public class Parent {
    public int m1(){ return 1; }
    public int m2(){ return 2; }
}
public interface Inter1
    { public int m3(); }
public class Child extends Parent implements Inter1 {
    public int m3() { return 3; }
    public int i = 4;
    public static void main(String[] args) {
        Child c = new Child();
        Inter1 i1 = c;
        Parent p = c;
        System.out.println(c.i+", "+i1.m3()+", "+p.m1());
    }
}
```


Casting

```
Parent p = new Child();
Child c = p;                    // Problem!

Child c2 = (Child)p;            // Problem solved!
```

- Casting rules:
  1. Subclass to superclass: no explicit cast required
  2. Superclass to subclass: explicit cast required
  3. Cannot cast unrelated types
  4. Object must be of the type being cast to or there will be a runtime error (`ClassCastException`)
  - `instanceof` can be used to determine compatibility at runtime

Virtual Methods

```
public class Employee {
   public double calcBonus(){
      double bonus;
      ...                  // Generic bonus calculation
      return bonus;
}
public class Manager extends Employee {
   public double calcBonus(){
      double bonus;
      ...                  // Specific bonus calculation
      return bonus;
}
Employee[] employees; // Contains Employees and Managers
for(e : employees) System.out.println(e.calcBonus());
```


Polymorphic Parameters

```
public void reviewBonus(Employee e) {
    System.out.println("Reviewing bonus for " +
          e.name + "Bonus = " + e.calcBonus());
}
```

- If the rules for method overriding were not enforced, polymorphism would not work
  - overridden class at least as accessible
  - no new or broader exceptions
  - covariant return types

Java SE8 OCA

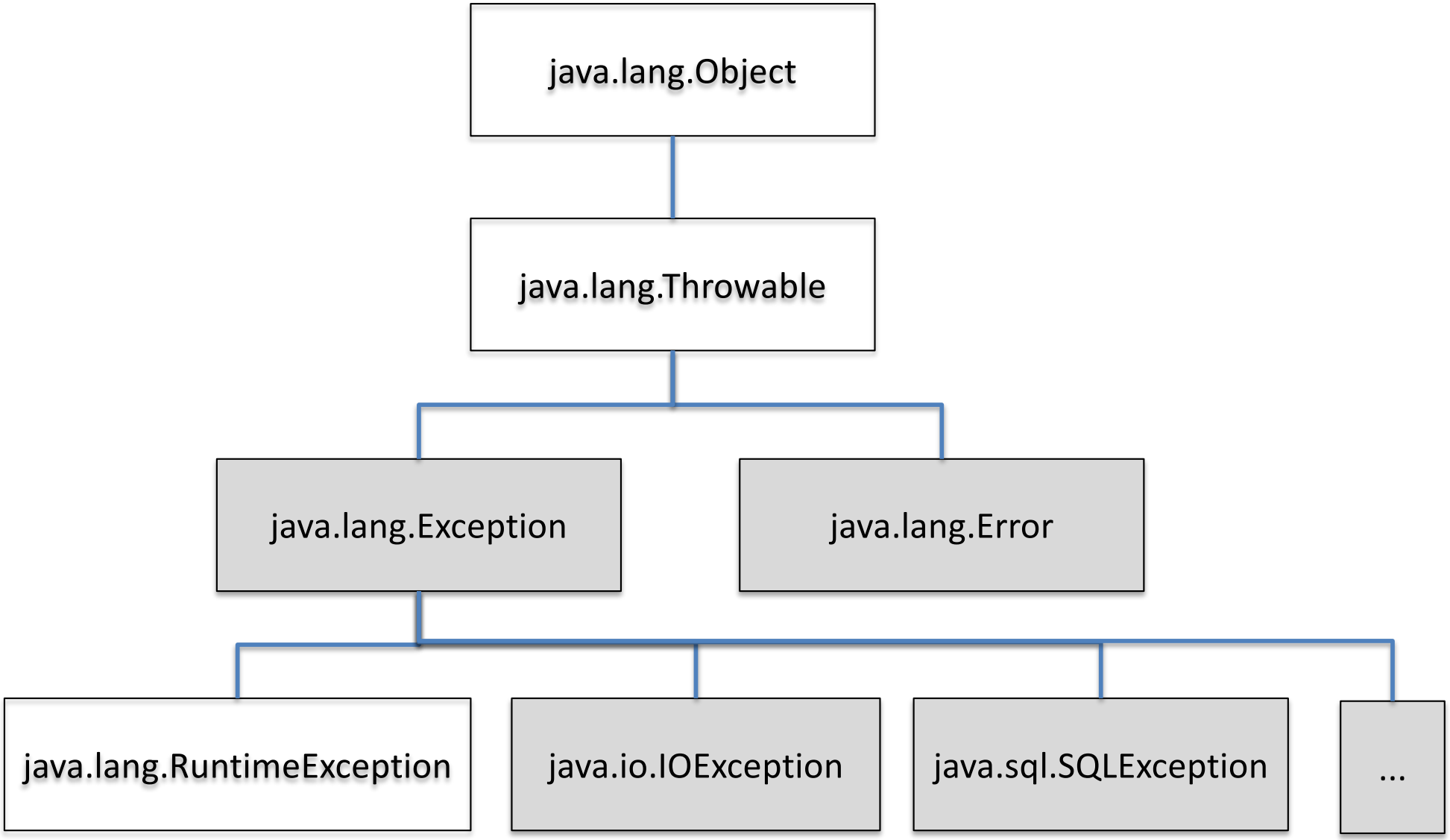
Session 6:
Exceptions

Exceptions at Runtime

```
public class Test {
    public static void main(String[] args) {
        int[] intArray = new int[3];
        for(int i = 0; i <= 3; i++)
            intArray[i] = i;
    }
}
```

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 3
    at Test.main(Test.java:5)
```


Exception Types



Handling Exceptions

```
try {
        // Code where an exception may be thrown
    } catch(exception_type e){
        // Code to handle the exception
    } catch(another_exception_type a) {
        ...
    }
        ...       // More catch blocks
    } finally {
        // Code to execute at the end
    }
}
```

- A try block must have one catch or finally block, may have both and may have many catch blocks

Catch Order

- Exceptions are classes
- Exceptions may be related – subclass/superclass
- Subclass exceptions will be caught by a superclass catch block

```
try {
    ...
    } catch (superException superE) {
    ...
    } catch (subException subE) {
    ...       // Code will never execute
    }
}
```


Throwing Exceptions

- JVM throws RuntimeExceptions:

```
int[] i = new int[1];
i[1] = 10; // throws ArrayIndexOutOfBoundsException
```

- Programmer can explicitly throw exceptions:

```
throw new Exception();
throw new RuntimeException("Message");
```

- Most exceptions accept an optional String parameter

Types of Exception

Type	Inheritance	Can be caught?	Required to handle?
Runtime exception	Subclass of RuntimeException	Yes	No
Checked exception	Subclass of Exception but not of RuntimeException	Yes	Yes
Error	Subclass of Error	No	No

Common Runtime Exceptions

- Thrown by JVM:
  - ArithmeticException
  - ArrayIndexOutOfBoundsException
  - ClassCastException
  - NullPointerException
- Thrown by programmer:
  - IllegalArgumentException
  - NumberFormatException (programmer)

Common Checked Exceptions

- FileNotFoundException
- IOException

Common Errors

- ExceptionInInitializerError
- StackOverflowError
- NoClassDefFoundError

Handling or Declaring Checked Exceptions

```
public class c1 {
    public static void main(String[] args) {
    m1();
    }
    public void m1() throws IOException {
    ...
    }
}
```

- Either need `main` to declare it `throws IOException`
- Or have a `try catch` block that handles `IOException`

Printing Exceptions

```
public static void main(String[] args) {
    try {
        m1();
    } catch (Exception e) {
        System.out.println(e);
        System.out.println(e.getMessage());
        e.printStackTrace();
    }
}
void m1() {
    throw new RuntimeException("Problem...");
}
```


Java SE8 OCA

Exam Prep:
Revision

Java Building Blocks

- Be able to write code using a main() method
- Understand the effect of using packages and imports
- Be able to recognise a constructor
- Be able to identify legal and illegal declarations and initialisation
- Be able to determine where variables go into and out of scope
- Be able to recognize misplaced statements in a class
- Know how to identify when an object is eligible for garbage collection

Operators and Statements

- Be able to write code that uses Java operators
- Be able to recognize which operators are associated with which data types
- Be able to write code that uses parentheses to override operator precedence
- Understand if and switch decision control statements
- Understand loop statements
- Understand how break and continue can change flow control

Core Java APIs

- Strings
- StringBuilder
- == and .equals()
- Arrays
- ArrayList
- Dates and Times

Methods and Encapsulation

- Method declarations
- Accessibility
- Static imports
- Static (class) v instance methods
- Constructors, default, this() and super()
- Encapsulation rules
- Lambda expressions

Class Design

- Extending classes
- Overriding
- Hiding (variables and statics)
- Overloading v. overriding
- Abstract classes
- Interfaces, default and static methods (Java 8)
- Polymorphism and virtual methods
- Casting reference variables

Exceptions

- Checked vs. unchecked
- try, catch, finally
- Programmer vs. JVM thrown exceptions
- Declaring and handling exceptions
- Throwing exceptions