# Java Style
# and Advanced Java

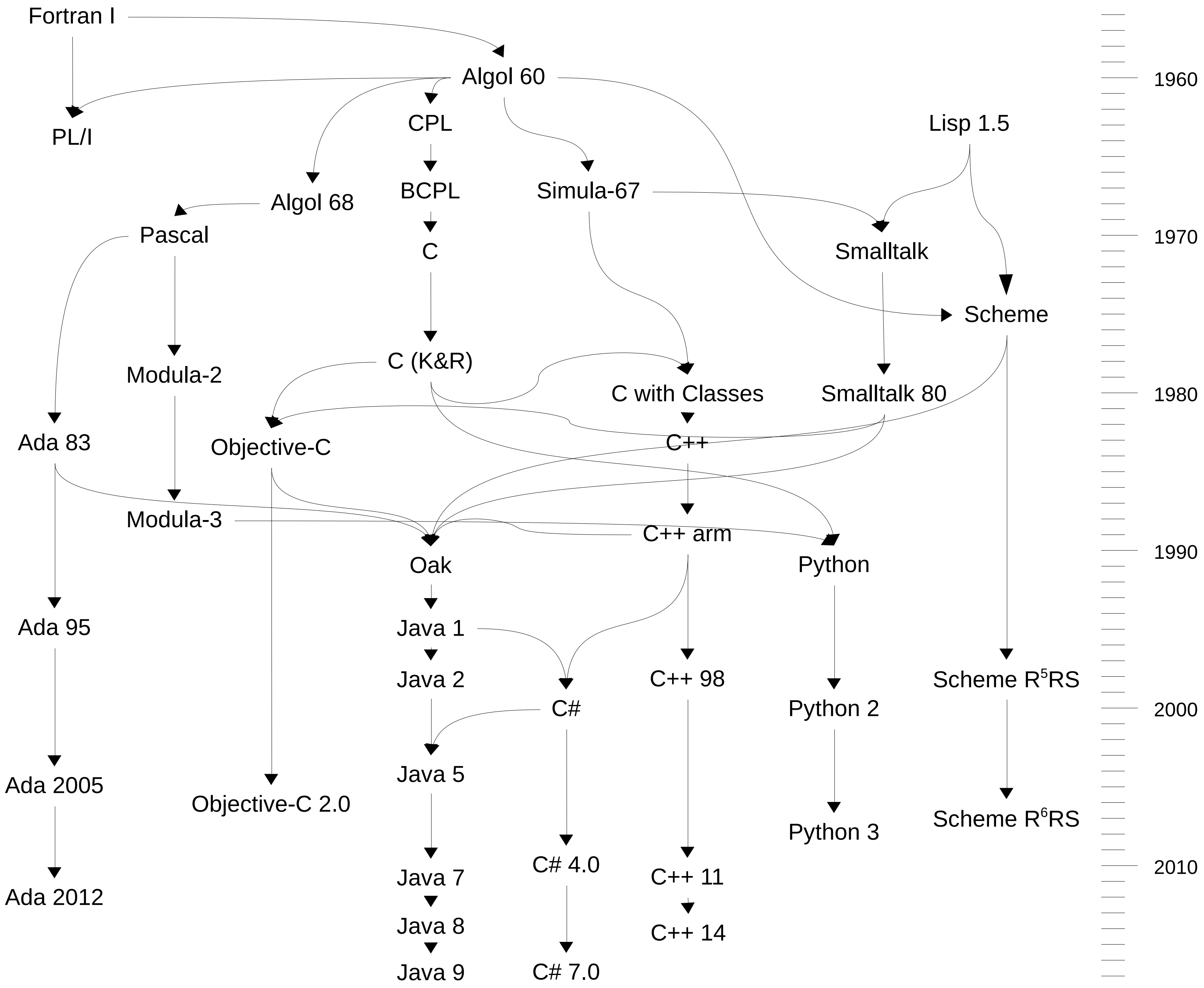
CS345 Winter 2018

# Outline

- Java Background
  - A little history and the JVM
  - Primitive versus Reference (Class) Types
  - Garbage Collection
- Java Style
- Structuring a Java program
- Miscellaneous topics

# A Little History

- Begun at Sun Microsystems in June 1991
  - Originally designed for interactive television
  - “Too advanced” for the time
- Released to public 1995
  - Java version 1.0
  - Idea: “Write Once, Run Anywhere”
    - Use Java to provide executable content for the Web
- Versions
  - Java 9 released September 21, 2017
    - Current: Java SE 9.0.4
  - Java 8 also known as Java SE 8
    - Current: Java SE 8u161
    - What we will be using for this course



Fortran I

Algol 60

1960

PL/I

CPL

Lisp 1.5

Algol 68

BCPL

Simula-67

Pascal

C

Smalltalk

1970

Scheme

Modula-2

C (K&R)

C with Classes

Smalltalk 80

1980

Ada 83

Objective-C

C++

Modula-3

C++ arm

Oak

Python

1990

Ada 95

Java 1

Java 2

C++ 98

Scheme R$^5$RS

C#

Python 2

2000

Java 5

Ada 2005

Objective-C 2.0

Scheme R$^6$RS

Python 3

C# 4.0

Java 7

C++ 11

2010

Ada 2012

Java 8

C++ 14

Java 9

C# 7.0

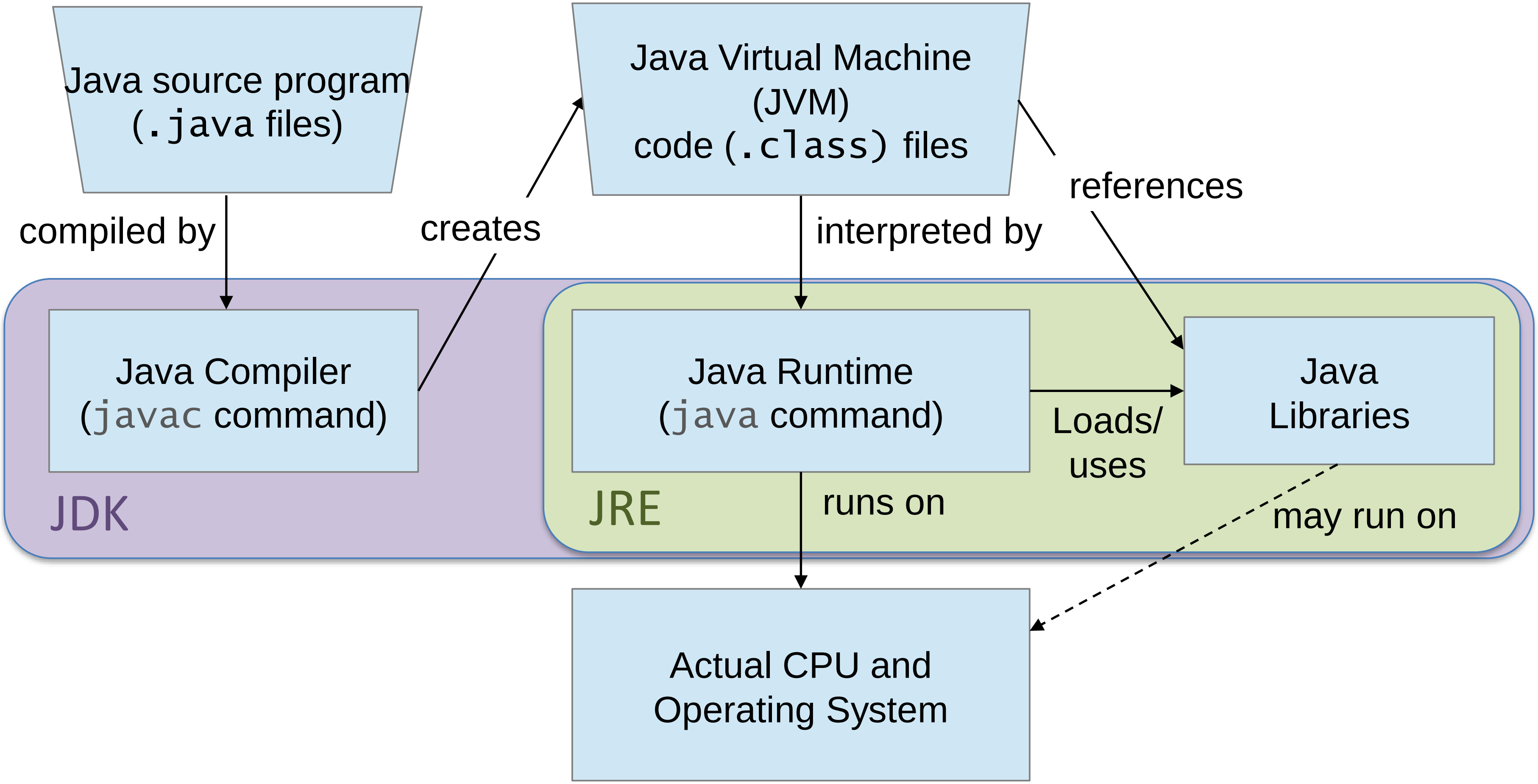
Source: Computer Languages History, Eric Levenez [http://www.levenez.com/lang/](http://www.levenez.com/lang/), down loaded April 3, 2015

# A Little History
# Goals of the Language

- Simple, object-oriented, and familiar
- Robust and secure
- Architecture-neutral and portable
  - “Write once, run anywhere!” You can build a Java application on one machine and run it on any other machine
    - Different CPU and/or different OS are OK
- High performance
- Interpreted, threaded and dynamic

# JVM/JRE/JDK (Highly simplified)

| | |
|---|---|
| JVM | Java Virtual Machine |
| JRE | Java Runtime Environment |
| JDK | Java Development Kit |



# Comment on JavaScript

- You would assume that JavaScript has something to do with Java.
  - You would be “mostly wrong.”
- JavaScript started in the early 1990s as LiveScript from Netscape Corp (now Mozilla).
  - Purpose: Provide active content in web pages.
- In 1995, when Java became hot, LiveScript was reworked to resemble Java and renamed JavaScript.
- In terms of programming, it's closer to Python than Java.
  - And, it's object model is not like either Python or Java.

# Outline

- Java Background
  - A little history and the JVM
  - ➡ Primitive versus Reference (Class) Types
  - Garbage Collection
- Java Style
- Structuring a Java program
- Miscellaneous topics

# Primitive and Reference Types

- Primitive Types
  - Complete list: `byte, short, int, long, char, float, double, boolean`
- Reference Types:
  - Everything else: objects and arrays
  - Objects (of a reference type) and all arrays are stored in the heap
  - All references are the same size

# In General

- Values of primitive types are stored directly
  - Local variables in the stack frame
  - Object attributes in the object
  - Array values in the array
- Objects and arrays are stored by reference
  - References for local variables in the stack frame
  - References for object attributes in the object
  - Arrays of reference types are arrays of references
    - A 2-D array of Strings is an array of references to arrays of references to Strings

# Value semantics

- *Value semantics*: Values (of primitive types) are copied when assigned to each other or passed as parameters.
  - Modifying the value of one variable does not affect others.

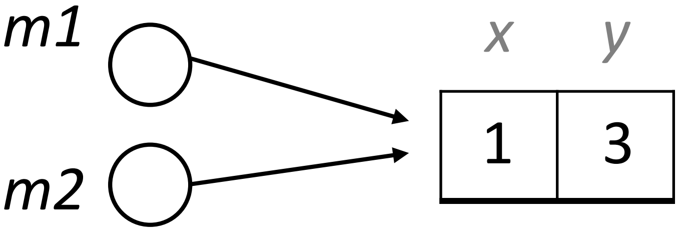
```
int x = 5;
int y = x;      // x = 5, y = 5
y = 17;         // x = 5, y = 17
x = 8;          // x = 8, y = 17
```

# Reference semantics

- *Reference semantics*: Variables store the reference to an object in the heap.
  - When one variable is assigned to another, the object is not copied; both variables refer to the same object.

```
class MyClass {int x; int y; }

MyClass m1 = new MyClass();
m1.x = 1; m1.y = 2;
MyClass m2 = m1;
m2.y = 3;
System.out.println(m1.y);   // 3
```



# Cautions

Be aware of reference semantics.

- Consider using *value types*.
  - A value type is one that “plays the role” of a primitive type even if it is a reference type
  - A value type is one where instances cannot be changed after being constructed.
  - `String` is a value type.

# Cautions

- == comparison works for:
  - primitive types
  - `enums`
  - some other special cases (**not** including Strings)

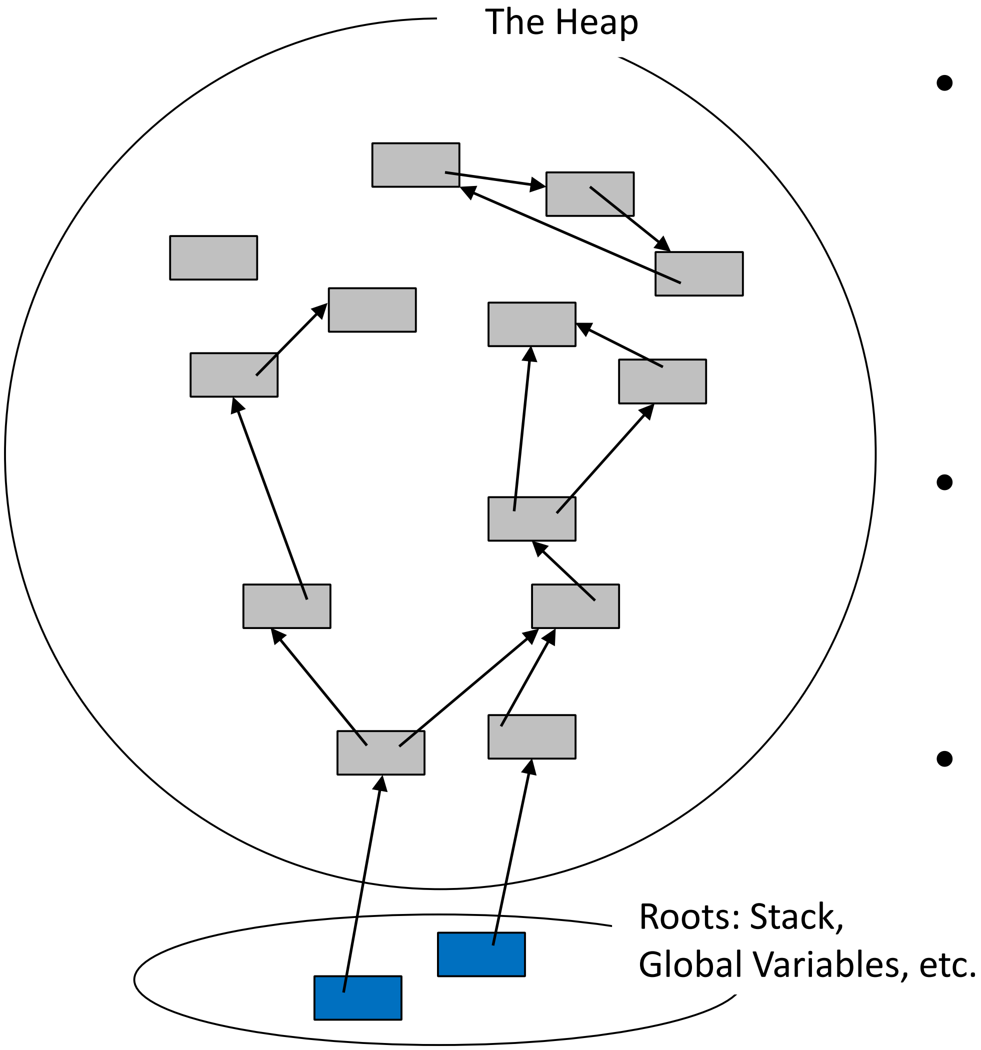
  Otherwise, use `.equals()`
  - use `s1.equals(s2)` to compare Strings for equality
    - This is true if the two strings are the same sequence of characters
  - `s1 == s2` is only true if `s1` and `s2` are the same String object

- Make a copy if / when you need to.
  - Use a “copy” constructor for objects.
  - Typical use-case: You are returning the value of a private, non-value type attribute that the caller cannot be allowed to modify. This is called a defensive copy!

# Outline

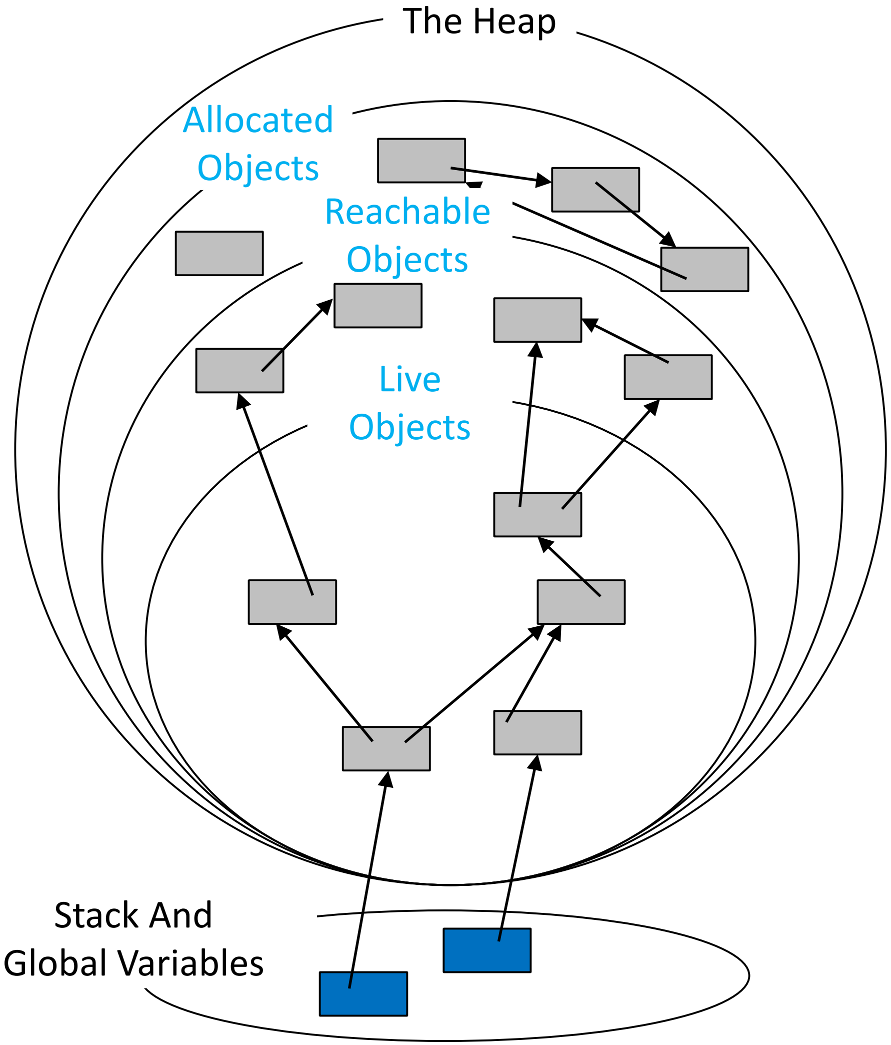
- Java Background
  - A little history and the JVM
  - Primitive versus Reference (Class) Types
  - Garbage Collection
- Java Style
- Structuring a Java program
- Miscellaneous topics

# Dynamic Memory (Java)



- The Stack and global variables can contain primitive types and references to objects of reference types.
- The data for all reference types are stored in the Heap.
- Objects stored in the Heap can contain references to other objects in the Heap.

# Terminology



- Allocated Objects – Objects allocated by the program and still in the heap

- Reachable Objects – Objects that are still reachable by following references from the the stack and global variables

- Live Objects – Objects the program might actually use in the future

# Some more terminology

- Syntactic garbage
  - Allocated objects that are not reachable
- Semantic garbage
  - Reachable objects that are not live

# Manual Memory Management

- C, C++, Ada (and others) use manual memory management

- In those languages, your program must explicitly deallocate (free, delete) memory for objects allocated on the heap.
  - Doing this correctly is hard.

# Manual Memory Management Correctness (1 of 2)

- Your program is correct as long as objects are deallocated:
  - After the object becomes semantic garbage
  - Before the object becomes syntactic garbage
    - Once an object becomes syntactic garbage, you no longer have a reference to it to use to deallocate it.

# Manual Memory Management Correctness (2 of 2)

- Error 1: Failing to free something before it becomes syntactic garbage.
  - Do enough of this and sooner or later you run out of memory.
  - This is known as a *memory leak*.

- Error 2: Freeing something that is still live.
  - Oh *&#*! Prepare for a strange and spectacular failure.
  - This is known as a *dangling pointer*.

# What is Garbage Collection (GC)?

- Automatic reclamation of syntactic garbage

- Why syntactic garbage?
  - Because a program can decide whether or not something is syntactic garbage.
  - In general, you can’t decide (It is mathematically, theoretically impossible!) whether or not something is semantic garbage.
    - This requires knowing all possible future behaviors of the program.

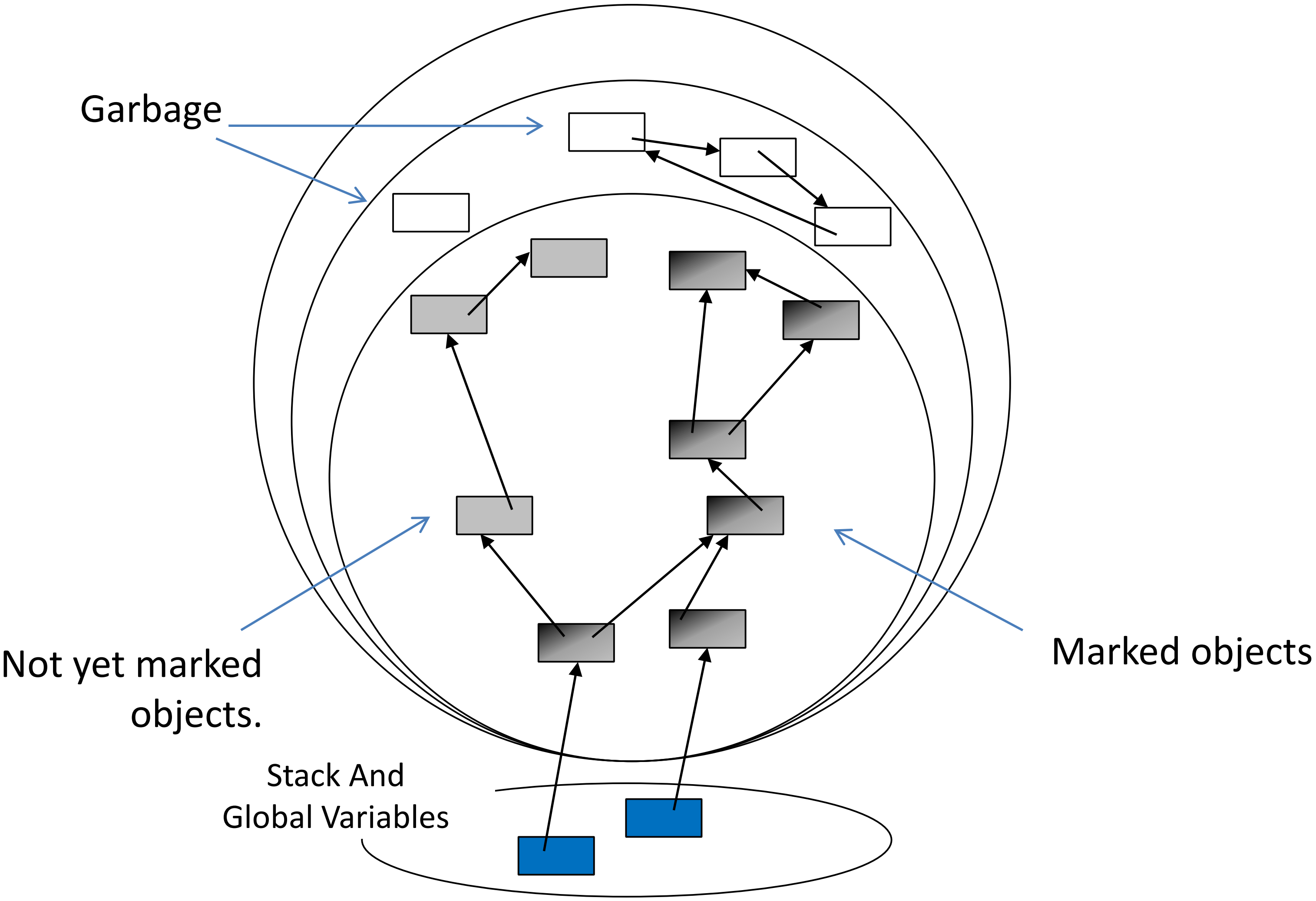
- Java uses GC
  - The JVM tracks everything allocated on the heap and knows how to trace all the references.

# How to you do GC?
# Basic Approach: Mark and Sweep

- Simple version:
  - Periodically, stop everything you are doing (*stop the world*).
  - Starting at the roots, recursively mark all objects that are reachable.
  - Anything that was not marked is garbage and is swept up and deallocated.

- Some well known users (maybe):
  - Java, C#, Go
  - Lisp/Scheme
  - Python to recover circular garbage

# Mark and Sweep



# GC Performance Measures

- Overhead
  - What percentage of CPU time is taken up by GC?
- Pause Time
  - How often and for how long does GC stop the application?
    - Important for web servers, user interfaces
- Stop-The-World
  - Does GC stop the application until GC is complete?

# What does GC cost? (1 of 2)

- Detlefs, Dosser, and Zorn, 1993
  - Compared an early version of  Boehm–Demers–Weiser conservative collector (a Mark and Sweep collector for C/C++) to various C language (malloc/free) implementations
  - Used applications that were designed for manual memory management.

- Result
  - Estimated average 21% increase in run time
  - Variation by application: -5% to 50% increase

- Criticisms
  - Collectors have improved significantly since then.
  - The applications were not designed for use with a garbage collector.
    - Many non-GC applications do additional copying that would not be required with a garbage collector.

# What does GC cost? (2 of 2)

- Hertz, Berger, 2005
  - Compared a number of Java applications
  - Compared differing garbage collectors (both Mark and Sweep and others) versus a (magic) oracle that knows when any object becomes semantic garbage.
- Results: Slow down versus Increased Heap Size
  - 70% slower at 2x minimum heap size
  - 17% slower at 3x minimum heap size
  - 1% slower at 5x minimum heap size
- Criticism
  - The oracle probably provides unrealistically good estimates for manual memory management.

# Why GC?

- Less buggy
  - In a large program, manual memory management is very, very (really very) hard to get right.
- Overhead is becoming less important
  - GC is questionable when overhead is important
- Life is too short
  - Accurate manual memory management requires careful design
    - And extra code
  - Chasing memory management bugs is painful and time consuming

# There is No Free Lunch!

- Princeton's driverless vehicle, entered in the 2005 DARPA Grand Challenge, was controlled by 10,000 lines of C# (a GC language much like Java).
- The vehicle ran for 28 minutes and 10 miles before succumbing to a "memory leak".

- Problem:
  - Obstacle objects were registered to listen for position updates.
  - Obstacles were "deleted" (dropped from consideration) once they were ten feet behind the vehicle
  - Obstacles were not removed as a listener for position updates– keeping them from becoming syntactic garbage. Oops!
  - A one line fix was found the day after the team lost the competition ($2 Million prize).

# Some Java Specifics

- Don’t leave references (pointers) lying around once you are done with them.
  - If you need to, set them to `null`.
    - Don’t bother unless the reference is <u>not</u> going to be overwritten or discarded in the immediate future.

- If you can, encapsulate object usage inside a single method.
  - References are created when they are needed and destroyed when you are done with them.

# Oracle Java 8 Specifics

| Java 8 has four possible collection algorithms: | Stop the World | Use Multiple Threads | Comments |
|---|---|---|---|
| Serial | Yes | No | For single threaded apps with small heaps |
| Parallel | Yes | Yes | Java 8 default. For apps that can tolerate pauses and want low overhead. |
| Concurrent Mark Sweep (CMS) | Usually no | Yes | Tries to avoid pauses with increased overhead. Intended for heaps < 4GB. |
| Garbage First (G1GC) | Usually no | Yes | Java 9 default. Divides heap in regions that are handled individually. Intended for heaps > 4GB. |

# Outline

- Java Background
- ➡ Java Style
  - Naming, comments, white space, indentation
  - Booleans
  - Generics
  - Collections
  - `for` loops
  - Exceptions
  - Enumerations
- Structuring a Java program
- Miscellaneous topics

# Style

- Professional software looks clean, neat, well-organized, etc.
  - All of which make it easier to read, test, debug, modify, enhance, etc..
- The following slides contain rules for assignments in this class.
  - Some of these are always good and some of these are my personal preferences that I am forcing you to use in this class.
  - I may add additional rules later.
- In general, be consistent!

# Outline

- Java Background
- Java Style
  - Naming, comments, white space, indentation
  - Booleans
  - Generics
  - Collections
  - `for` loops
  - Exceptions
  - Enumerations
- Structuring a Java program
- Miscellaneous topics

# Style
# Naming

- Use names that are short but meaningful in context.
- For example:

```
for (int i = 0; i < allWords.length; i++) { … }
```

  - is fine if the body of the loop is small. (`i` has small scope.) This is questionable if the loop body is large.
  - If the body is large, think about names like `wordIndex`, `wordNum`, `curWord`, etc.

# Style
# Naming

- Top-level class and interface names and public attribute and method names must be descriptive in some global sense.
  - The context for these names is the whole program.

- Never use as a name: `foo`, `bar`, your friend's name, etc.

- Multiple names like `x1`, `x2`, `x3`, … are suspect.

# Style
# Naming

- Use the following (Java standard) naming conventions:

| What | Naming Standard |
|---|---|
| package | `alllowercase` |
| class, enum, interface | `UpperCaseCamelCase` |
| method, attribute, variable, parameter | `lowerCaseCamelCase` |
| constant | `ALL_UPPER_CASE` |

# Style
# Comments

- Use comments at the start of every`.java` file to identify:
  - The problem/assignment
  - The author (you)
  - Put these before any `package` or `import` statements.

- Use comments at the start of each class to identify the purpose and/or responsibilities of the class.
  - Put these before the `class` statement

- Use comments at the start of each non-obvious, non-private method and constructor to describe:
  - The purpose of the method.
  - Each of the parameters.
  - What is returned (for non-void methods)
  - Exceptions that can be thrown.
  - Put these before the method or constructor.

# Style
# Comments Example

```
/* Comment Example
 * Author: Chris Reedy
 * CS345 Winter 2018 */
package mypackage;

/* Example class
 * Demonstrates comments */
class Example {

    /* Return the value associated
     * with this Example and count number
     * of calls. */
    int getValue() {
        getValueCalls += 1;
        return value;
    }
}
```

# Style
# Comments

- For long stretches of code, provide comments that help to identify the purpose of sections of the code. For example, “Find the Word object associated with the user’s input.”
  - Or, break the code into multiple small methods. Name the methods to show what the method is intended to do.

- Don’t restate what’s obvious from looking at the code.
  - No:

    ```
    x += 1; // Add one to x
    ```
  - Some possible good reasons for making a comment here:
    - Why are we adding one and not something else.
    - This has to be done now and not earlier or later.

# Style
# Comments

- Java has two kinds of comments:

```
// Single line comments

/* Comments between slash-star
 * and star-slash. These can
 * extend over multiple lines. */
```

  – In general:
    - Use // style for single line comments
    - Use /*  …  */ style for multi-line “block” comments
    - Use /*  …  */ style for documentation comments
    - Use // style to comment out code
      – This will comment out /*  …  */ and // style comments

# Style
# Comments

- // versus /* … */ usage is stylistic, not mandatory.
  - However, be consistent.

- Comments that “comment out” debugging code are acceptable.
  - Remember to comment out all debugging output before submitting your program.
  - All (?) Java editors have a command that will comment or un-comment a block of code.

- In general, use the following standard: comments should provide information you would want to know if you were looking at this code for the first time.

# Style
# White Space

- Use a blank line(s) to separate
  - Methods, constructors, classes from each other.
  - Methods, constructors, classes from attributes.
  - Groups of attributes if there are a lot of attributes.

- Never use three consecutive blank lines.
  - In general, use only one blank line. If you find yourself wanting to use two, do you need a comment instead?

- In long blocks of code, use a blank line to separate sections of the code.

# Style
# Indentation

- You must indent your program. The indentation should reflect the block structure.
  - Indent twice to avoid confusion between continuation of a single statement and the start of a new statement. For example,
    - No

```
if (someCondition ||
    someOtherCondition ||
    yetAnotherCondition) {
    doSomething();
}
```

- Yes

```
if (someCondition ||
        someOtherCondition ||
        yetAnotherCondition) {
    doSomething();
}
```

# Style
# Indentation

- Use 2, 3, or 4 spaces for indentation.
  - Do not use tabs!
  - Use the same indentation throughout a single .java file.
  - Find out what your editor uses and how to change it.
  - Set your editor to use “soft tabs”—tab characters are converted to spaces.
  - The big Java IDEs, Eclipse, IntelliJ, … can format your code.
  - This is my personal preference.
    - And, your program looks the same in all editors.

- Keep all lines under 100 characters
  - I try to keep mine under 80 characters
  - I hate horizontal scrolling!

# Style
# Indentation and Braces

- Braces for statements with blocks (if, while, for, try, …) can be:

```
while (aCondition) {
    . . .
}
```

or

```
while (aCondition)
{
    . . .
}
```

  - The first is the standard at both Google and Oracle and my personal preference. This is called “Egyptian brackets”.

- Whatever you choose, you must be consistent.

# Style
# Indentation

- If you have a block with a single statement:
  - Acceptable:

```
if (condition)
    doSomething();
```

  - Preferred:

```
if (condition) {
    doSomething();
}
```

  - No:

```
if (condition) doSomething();
```

  - The first option is error-prone:

```
if (condition)
    doSomething();
    doSomethingElse();
```

    - `doSomethingElse()` is not protected by the if statement!

# Outline

- Java Background
- Java Style
  - Naming, comments, white space, indentation
  - Booleans
  - Generics
  - Collections
  - for loops
  - Exceptions
  - Enumerations
- Structuring a Java program
- Miscellaneous topics

# Style
# Booleans

- No

```
if (aBoolean == true)
    ...
if (aBoolean == false)
    ...
```

- Yes

```
if (aBoolean)
    ...
if (!aBoolean)
    ...
```

- No

```
if (aCondition)
    return true;
else
    return false;
```

Yes

```
return aCondition;
```

# Outline

- Java Background
- Java Style
  - Naming, comments, white space, indentation
  - Booleans
  - Generics
  - Collections
  - for loops
  - Exceptions
  - Enumerations
- Structuring a Java program
- Miscellaneous topics

# Generics

- Generics allow you to write classes, interfaces and methods that are parameterized (can vary) based on specific types (classes and interfaces)

# Example: Generic List

- Without generics (1)

```
List l = new ArrayList(); // List of Strings
l.add("Hello");
String s = (String)l.get(0); // Cast to String
```

- Without generics (2)

```
List l = new ArrayList(); // List of Strings
l.add(5); // Wrong but allowed!
String s = (String)l.get(0); // ClassCastException!
```

- With generics

```
List<String> l = new ArrayList<>(); // List of Strings
l.add("Hello");
l.add(5); // Compilation error
String s = l.get(0); // No cast needed
```

# A Generic Class

- Example: A Box class that “boxes” an arbitrary reference type

```
public class Box<T> {
    // T stands for "Type”
    private T t;
    public void set(T t) {
        this.t = t; }
    public T get() {
        return t; }
}
```

# Generic Naming Conventions

- T, S, U, V – first, second, etc. type names
- E – element for collections
- K, V – key and value for maps
- N – number

# Using Generic Classes

- Declaring a variable, etc.

```
    Box<Integer> box;
```

- Creating an instance (1)

```
    Box<Integer> box = new Box<Integer>();
```

- Creating an instance (2)

```
    Box<Integer> box = new Box<>();
```

  – Compiler infers the generic type

# Raw Types

- Consider

```
Box box;
```

- Box is referred to as a *raw type*
  - Box is actually `Box<Object>`
  - Style: Don’t use raw types
  - Compiler will warn about raw types with correct flags
    - `javac –Xlint:rawtypes`
    - IDEs: see compiler flags settings to set this

# Generic Methods

- From the standard List interface

  ```
  <T> T[] toArray(T[] a);
  ```

  - T is a type
  - Returns an array of type T[] with the contents of the list. Returns a if the list fits in a. Otherwise, returns a new array with the type T[].

# Generic Method Usage

- Usage:

```
String[] y = x.toArray(new String[0]);
```

  - The compiler infers that T is `String`
  - Complete rules for type inferencing are messy
    - If you're unsure, try it. The compiler will object if it can't.

- Usage—explicit specification:

```
String[] y = x.<String>toArray(…);
```

- Style: use of type inferencing preferred

# Bounded Parameters

- Messy example:

```
public static <T extends Comparable<T>>
    int countGreaterThan(T[] anArray, T elem)
{
    int count = 0;
    for (T e : anArray)
        if (e.compareTo(elem) > 0)
            ++count;
    return count;
}
```

- “extends” says that the type T must extend or implement the named type.
- The interface `Comparable<T>` provides the `compareTo` method needed in the body.

# Wildcard Parameters

- Unbounded wildcard—`list` is a list of anything

```
public static void printList(List<?> list) {
    for (Object elem: list)
        System.out.print(elem + " ");
    System.out.println();
}
```

- Upper bounded wildcard—`list` is a list of Foo or a subclass (Foo a class) or an implementing class or extending interface (Foo an interface)

```
public static void process(List<? extends Foo> list) {
    for (Foo elem : list) {
        // ...    }
}
```

- Lower bounded wildcard—`list` is a list of a superclass of Integer or an interface implemented by Integer

```
public static void addNumbers(List<? super Integer> list) {
    for (int i = 1; i <= 10; i++) {
        list.add(i);
    }
}
```

# Outline

- Java Background
- Java Style
  - Naming, comments, white space, indentation
  - Booleans
  - Generics
  - ➡ Collections
  - `for` loops
  - Exceptions
  - Enumerations
- Structuring a Java program
- Miscellaneous topics

# Collections

- Java provides a complete set of collection (lists, maps, sets, etc.) interfaces and classes
  - All Java standard collection interfaces and classes are in the package `java.util`.
  - Style: you are free to use `import java.util.*;` to get all the collection types.

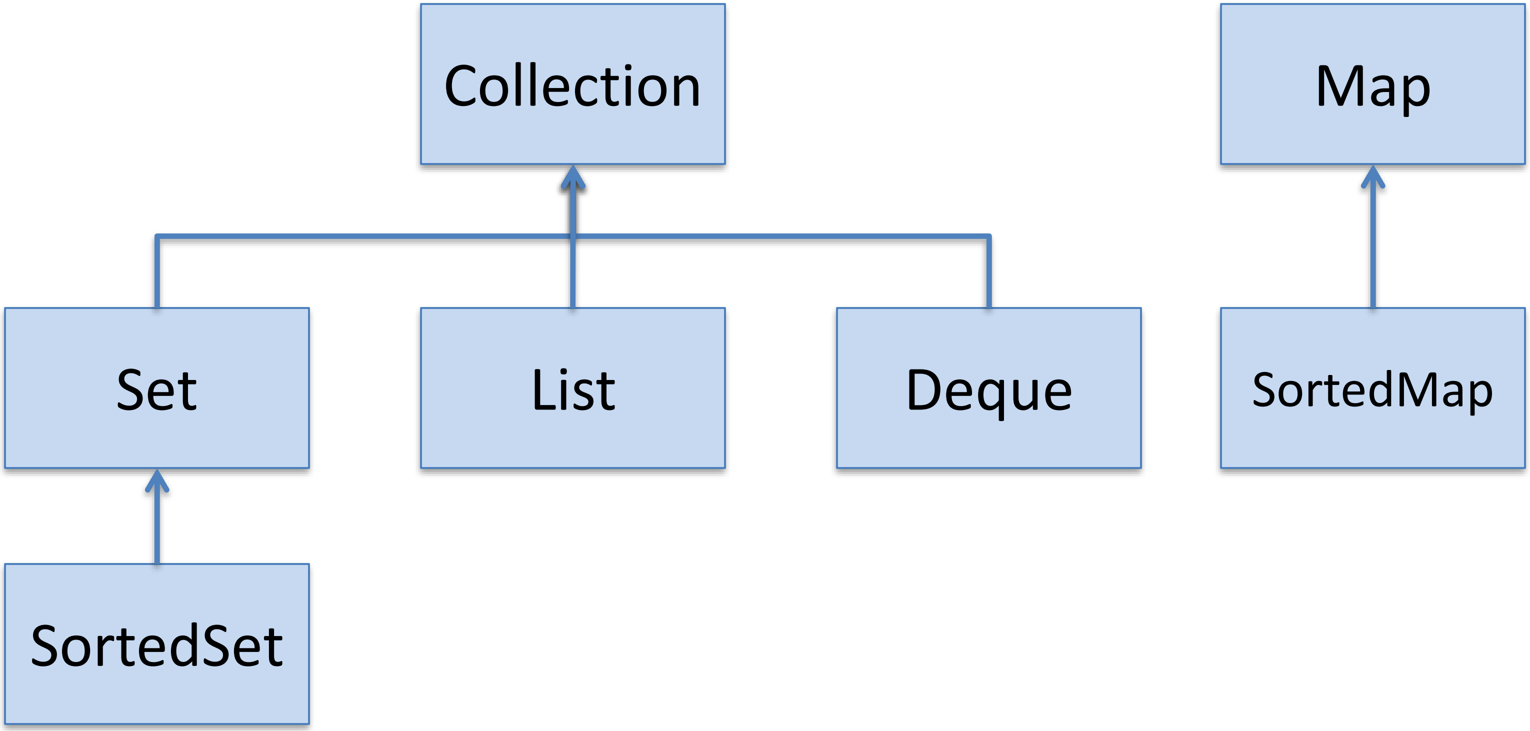
- Java collections are all generic:

```
List<MyClass> myList = new ArrayList<>();
myList.add(anObject);
```

  - `anObject` must be of type `MyClass` or a subtype

# Collections
# Interfaces

- Here are the important collection **interfaces** (Abstract Data Types)



# Collections
# Interfaces

- *Collection* – a collection of elements. Can be ordered or unordered. May or may not allow duplicates.
- *Set* – an unordered collection of elements that does not allow duplicates.
- *List* – an ordered collection. Duplicates are allowed. Also known as a *sequence*.
- *Deque* – a “double ended queue.” A combined stack and queue. (Also like a *deck* of cards.)
- *Map* – a map from keys to values. Duplicate keys are not allowed. Hash tables are maps.
- *SortedSet, SortedMap* – A Set or Map maintained in sorted order. Binary search trees are sorted maps.

# Collections Implementations

- Implementations are concrete classes that implement interfaces.
- Here are the important implementations:

| Interface | Hash Table | Resizable Array | Linked List | Hash table + Linked List | Tree |
|---|---|---|---|---|---|
| Set | HashSet | | | LinkedHashSet | TreeSet |
| List | | ArrayList | LinkedList | | |
| Deque | | ArrayDeque | LinkedList | | |
| Map | HashMap | | | LinkedHashMap | TreeMap |
| SortedSet | | | | | TreeSet |
| SortedMap | | | | | TreeMap |

# Using Collections
# Iteration

- To iterate through the members of a Collection:

```
List<MyClass> myList;
...
for (MyClass aMyClass : myList) {
    // do something with aMyClass
    // for example
    System.out.println(aMyClass);
}
```

  - `aMyClass` is a new local variable that has a scope of the body of the loop

# Using Collections
# Modifying Collections (1 of 2)

- In general, you can’t modify a collection while iterating through the members:

```
for (Etype e : myCollection) {
     if (…)
          myCollection.remove(e);
}
```

  - Will result in a `ConcurrentModificationException`.

# Using Collections
# Modifying Collections (2 of 2)

- To remove elements from a collection while iterating do this:

```
for (Iterator<Etype> iter =
        myCollection.iterator();
        iter.hasNext();) {
    Etype e = iter.next();
    if (…)
        // Remove current element
        iter.remove();
}
```

# Collections Implementations Notes

- LinkedHashSet and LinkedHashMap maintain elements in the order they are added to the collection.
  - Elements have a predetermined order when iterating through the Set or Map.
    - Not true for HashSet or HashMap.

# Collections Implementations Notes

- Check the Java library documentation for details of exactly how all the collections work.
  - Documentation for `java.util.*`: `http://docs.oracle.com/javase/8/docs/api/java/util/package-summary.html`

# Collections and Reference Types

- All collections (except arrays, which are handled separately by the JVM) are collections of reference types.
  - You can’t do `List<int>`
  - You can do `List<Integer>`
- All primitives types have corresponding reference, value type classes that “box” the primitive type:

| Primitive | Reference | Primitive | Reference |
|---|---|---|---|
| byte | Byte | char | Character |
| short | Short | float | Float |
| int | Integer | double | Double |
| long | Long | boolean | Boolean |

# Collections and Reference Types

- Additional notes:
  - All of the reference classes (`Integer`, etc.) are in the package `java.lang`. (No `import` required.)
  - The reference classes have a bunch of other useful stuff (see documentation). For example,
    - `Integer.MAX_VALUE` gives the largest possible `int`.
    - `Integer.parseInt(String s)` returns the `int` given by the `String s`.
      - `Integer.valueOf(s)` returns an `Integer`.
  - In general, the Java compiler will automatically convert between primitive types and their corresponding reference types, for example, between `int`s and `Integer`s. The following works:

```
List<Integer> myList = ... ;
myList.add(5);
int x = myList.get(0); // x == 5
```

# Collections
# Style Rules

- In your assignments in this class, don’t build your own collections (linked list, hash table, etc.). Use one from the Java library, specifically the `java.util` package.
  - Exception: You can build your own collection if what you need is not in the standard library.

# Collections
# Style Rules

- Local variables, attributes, parameters, or returned values that are collections should have a type that is one of the collections interfaces, not one of the implementations.
  - No:

    ```
    LinkedList<String> myList = new LinkedList<>();
    ArrayList<String> buildList(ArrayList<String> data) {
         … }
    ```

  - Yes:

    ```
    List<String> myList = new LinkedList<>();
    List<String> buildList(List<String> data) { … }
    ```

# Collections
# Style Rules

- Use the least restrictive collections interface possible.
  - Don't do:

    ```
    void f(List<String> data) { … }
    ```

    If you don't expect the elements of the collection to have a particular order. Instead use:

    ```
    void f(Collection<String> data) { … }
    ```

- Use the most restrictive generic type possible.
  - Don't do:

    ```
    List<Object> myList = … ;
    ```

    If myList will never hold anything but Strings (or some other reference type).

# Collections
# Style Rules

- Never use *raw* types or *unchecked* conversions:
  - No:

```
List myList;
```

  - or

```
myList = new ArrayList();
```

  - or

```
List<String> myStrings;
List myList = myStrings; // Legal!
```

# Outline

- Java Background
- Java Style
  - Naming, comments, white space, indentation
  - Booleans
  - Generics
  - Collections
  - `for` loops
  - Exceptions
  - Enumerations
- Structuring a Java program
- Miscellaneous topics

# Style
# for loops

- Use the *for-each (enhanced for)* statement when you can:
  - No

    ```
    List<String> myList;
    for (int i = 0; i < myList.size(); i++) {
        System.out.println(myList.get(i));
    }
    ```

  - Yes

    ```
    List<String> myList;
    for (String s : myList) {
        System.out.println(s);
    }
    ```

  - Yes—applies to arrays as well

    ```
    String[] myStringArray;
    for (String s : myStringArray) {
        System.out.println(s);
    }
    ```

# Style
# for loops

- Common reasons for not using for-each:
  - You need to modify the underlying collection in some way
  - For arrays, you need the array index to modify the array
  - You need to iterate through multiple collections “at once”
    - Example: Merging two sorted lists (merge sort)

# Style
# for loops

- When manually iterating through a collection, use the iterator:
  - No

    ```
    for (int i = 0; i < myList.size(); i++) {
        System.out.println(myList.get(i));
    }
    ```
  - Yes

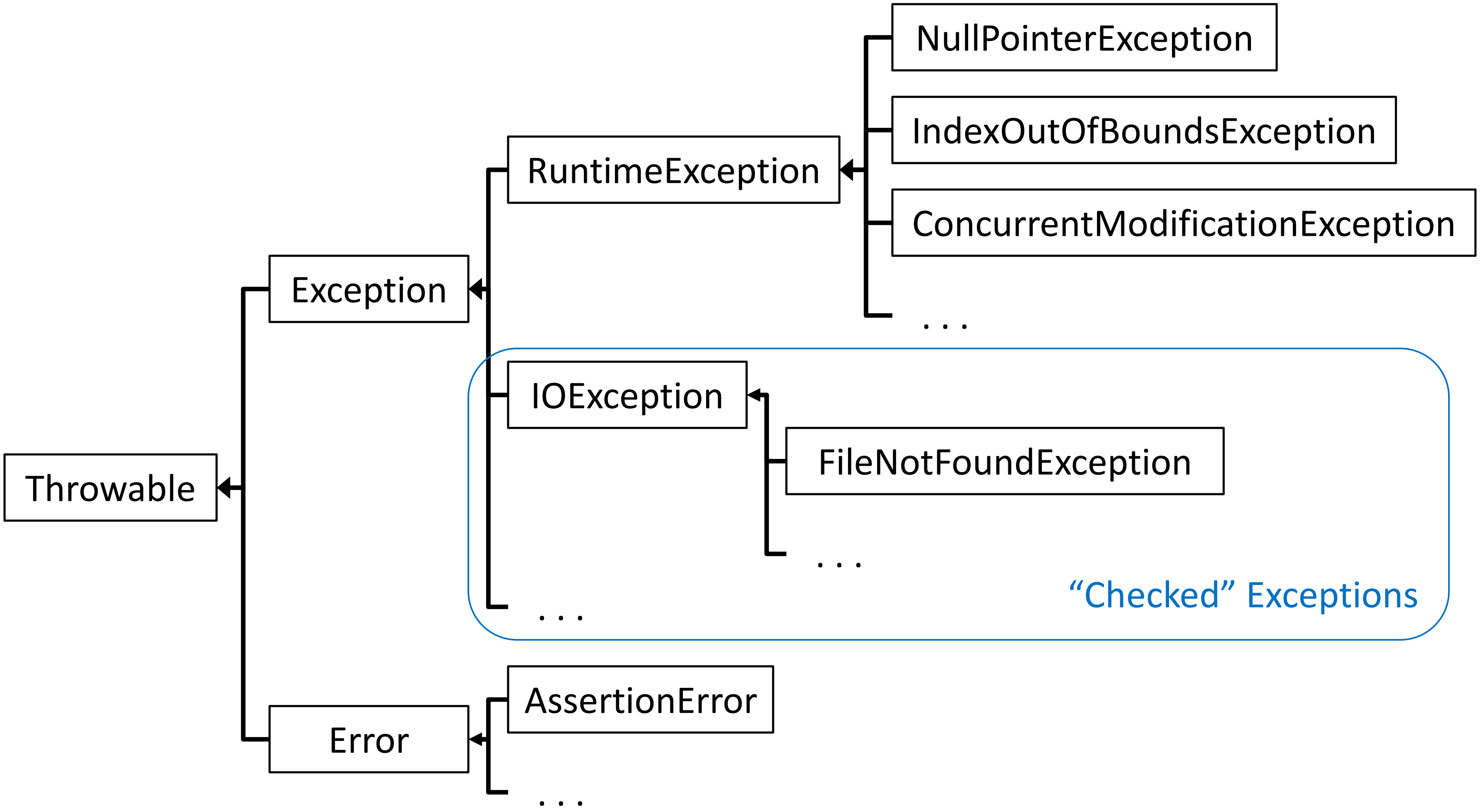
    ```
    for (Iterator<String> iter = myList.iterator();
            iter.hasNext(); ) {
        String elt = iter.next();
        System.out.println(elt);
    }
    ```

- If you need more flexibility (iterating backward, adding, removing, or setting elements) see documentation for `ListIterator`.
  - Note: You can only use `ListIterator`s on `List`s.
    - Backward has no meaning if the elements have no order.

# Outline

- Java Background
- Java Style
  - Naming, comments, white space, indentation
  - Booleans
  - Generics
  - Collections
  - `for` loops
  - Exceptions
  - Enumerations
- Structuring a Java program
- Miscellaneous topics

# Exceptions
# Exception Class Inheritance Hierarchy



# Exceptions

- Java Exceptions are classed as “checked” or “unchecked”.
  - Unchecked exceptions are Exceptions other than checked.
- Philosophy:
  - Checked exceptions are ones the program could handle and/or recover from.
  - Unchecked exceptions represent “bugs”.
- Methods that can throw a checked exception must say so.
- Example:

```
InputStream getInStream(String name)
        throws FileNotFoundException {
    // Can throw FileNotFoundException
    return new FileInputStream(name);
}
```

# Style
# Exceptions

- Never catch any of:
  - `Throwable`, `Error`, `Exception`, `RuntimeException`, `NullPointerException`, `IndexOutOfBoundsException` , `ArrayIndexOutOfBoundsException`, `ConcurrentModificationException`, `ClassCastException`, or any kind of `Error` (such as `AssertionError`).

# Style
# Exceptions

- If you catch an exception you cannot ignore it.
  - No

```
try {
    ...
} catch (NumberFormatException ex) { }
```

  - Yes

```
try {
    ...
} catch (NumberFormatException ex) {
    // Say that things didn’t work.
    System.out.println("An error message");
    // Or do some other fix up.
}
```

# Style
# Exceptions

- Explanation: The purpose of these rules is to prevent you from hiding problems by catching exceptions and ignoring them.
- Escape clause: If you’re convinced that violating one of these rules is the right thing to do, you must comment your code providing the justification. For example:

```
try {
    port = Integer.parseInt(s);
} catch (NumberFormatException ex) {
    /* If there is a NumberFormatException
       the value of port will be the
       unchanged default value of 80. */
}
```

# Style
# Exceptions

- The following is allowed/encouraged:

```
public static void main(String[] args) throws Exception
    // Says that main can throw any checked exception
    // in addition to a RuntimeException or Error
{
```

- If the compiler insists that you do something about a checked exception and, as far as you’re concerned, that exception is a bug, convert the checked exception into an AssertionError:

```
try {
    inStream = new FileInputStream(name);
} catch (FileNotFoundException ex) {
    // Treat file not found as a bug!
    throw new AssertionError(
        "Unexpected missing file”, ex);
}
```

# Outline

- Java Background
- Java Style
  - Naming, comments, white space, indentation
  - Booleans
  - Generics
  - Collections
  - `for` loops
  - Exceptions
  - Enumerations
- Structuring a Java program
- Miscellaneous topics

# Why Enumerations?

How do you create a predetermined, named set of objects?

- For example, days of the week

- `int day = 1;`
  - Is the day Sunday, Monday, or Tuesday? Better is:

    -
      ```
      static int MONDAY = 1; // Define a constant
      int day = MONDAY; // Use the constant
      ```
  - Is the first day zero or one? (Sunday or Monday)
  - What if day `== -1`?
- `String day = "Monday";`
  - Is that spelled “Monday”, “monday”, “MONDAY”?
    - Compiler won’t tell you
  - You have to remember to compare with .equals()
    - `if (day.equals("Monday")) …` , not
    - `if (day == "Monday") …`

# Enumerations

- When you want a predetermined, named set of instances, use an *enum*.

- Example:

```
public enum DayOfWeek {
    MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY,
    SUNDAY }
```

# Enumerations

- `enums` are classes with a guaranteed, predetermined set of instances. The names of the instances are static constants of the class.
  - `DayOfWeek.TUESDAY` is that particular instance.
  - Note: That's why the names are ALL_CAPS.

- Comparison with == works because there is never more than one instance of a specific enumeration:
  - `if (day == DayOfWeek.MONDAY) …`

# Enumerations

- Some useful methods (`E` is an enum type, `e` is an instance of `E`)
  - `e.toString()` – returns the name of `e` (e.g. “TUESDAY”)
  - `e.ordinal()` – returns the ordinal number of `e`, starting at zero
  - `E.valueOf(String s)` – returns the instance whose name is `s`
  - `E.values()` – returns an array of all the instances of `E`

# Enumerations

- Enumerations are “first class” classes
  - They can have attributes, methods, constructors, etc.
  - For example:

```
enum Coin {
    PENNY(1), NICKEL(5), DIME(10), QUARTER(25);

    private final int value;

    Coin(int value) { this.value = value; }

    public int getValue() { return value; }
}
```

  - Note: The constructor is automatically private.

# Outline

- Java Background
- Java Style
- Structuring a Java program
  - Packages and `import`
  - Accessibility (`public`, `private`, etc.)
  - Style Rules
- Miscellaneous topics

# Packages and `import` Outline

- Putting classes in packages
- What’s in a `.java` file?
- `import`
- Organizing classes and packages in directories
- Miscellaneous

# Packages and `import`
# Putting Classes in Packages

- The package statement is the first non-blank, non-comment statement in a `.java` file:

```
// header comment
package mypackage;

import … ;

// class definition(s)
class MyClass { ... }
```

Comment(s) covering author, copyright, assignment number, etc.

Comment(s) describing the class.

# Packages and `import`
# Putting Classes in a Package

- Packages may be “nested” using dotted notation:

```
package mypackage.sub1.sub2;
```

- The *fully qualified* name of a type (class or interface or enum) is the package name followed by the class name:

```
mypackage.sub1.sub2.MyClass
```

- The *fully qualified* name of a static member of a type is the package name followed by the class name and member name.

```
mypackage.sub1.sub2.MyClass.CONSTANT
```

- Types with no package statement are placed in the *“default package”*.

# Packages and `import`
# Naming Packages

- By convention, package names are all lower case:
  `java, java.util, mypackage`

- To avoid conflicts, convention dictates that organizations use reverse, dotted, Internet names for their packages. For example:
  `org.w3c.dom`
  `edu.wwu.cs.reedyc2.cs345.winter2018`
  - We won’t be doing this. (Whew!)

- The package names `java` and `javax` are reserved for the Java language

# Packages and `import`
# What's in a `.java` File?

- The compiler allows you to have multiple top-level types (classes and interfaces and enums) in a `.java` file.
  - Each type will have it's own `.class` file.
- At most one type in the file can be `public`, and it must have the same name as the file.
- In general, the only type that should be referred to outside of a given `.java` file is the type with the same name as the file.

# Packages and `import`

# What's in a `.java` File?

Example: `MyClass.java`:

```
package mypackage;
[public] class MyClass { ... }
class Helper { ... }
```

top-level classes

- Code outside of `MyClass.java` should only refer to `MyClass`.
- `MyClass` and `Helper` can refer to each other.
- `MyClass` can be public (it doesn't have to be).
- `Helper` cannot be public (or private or protected).
- No other classes in `mypackage` can be named `Helper`.
- The compiler will generate separate `MyClass.class` and `Helper.class` files.

# Packages and import
# import (Using Classes)

- You can always use a type or a static member (no import needed) by giving it’s fully qualified name:

```
java.util.Scanner input =
    new java.util.Scanner(System.in);
```

- Importing a type, allows you to refer to that type by its simple, unqualified name:

```
import java.util.Scanner;
Scanner input = new Scanner(System.in);
```

  – import statements can only occur after the package statement, if any, and before the first top level class definition

# Packages and `import`
# `import` statements

- To import a single type:
  ```
  import java.util.Scanner;
  ```
  - `import` always uses the fully qualified name.

- To *on-demand* import all types in a package:
  ```
  import java.util.*;
  ```

- To import a single static member of a type:
  ```
  import static java.lang.Math.sqrt;
  ```
  - You can use the `sqrt` function (a static function in class `Math`) with no further qualification, as opposed to `Math.sqrt`.

- To *on-demand* import all static members of a type:
  ```
  import static java.lang.Math.*;
  ```

# Packages and import
# import

- It is an error if a **single** import statement provides multiple definitions for the same name.

- import *package*.* will only import classes when they are needed to provide a definition for an otherwise unknown symbol.
  - It is not an error for on-demand import statements to provide multiple potential definitions for the same name.
  - Single import statements take precedence over on-demand import statements.
  - It is an error if a name is used that has multiple potential on-demand definitions.

# Packages and `import`

# `import` is not always required

- The package `java.lang` is implicitly (automatically) imported by the compiler, as if your program included:

  ```
  import java.lang.*;
  ```

  - `String`, `StringBuilder`, `Math` , `Integer`, `Exception`, and many others, are in `java.lang`.

- All types in a package are implicitly imported into types in the same package.
  - This includes classes in the “default” package
    - You cannot otherwise import classes in the default package.

# Organizing Classes and Packages in the File System

- Java expects type to be in files with the same name as the type.
  - Source for `class MyClass` is in `MyClass.java`
  - JVM code for `class MyClass` is in `MyClass.class`

- Packages are organized into corresponding directories in the file system. For example, for a class `mypackage.sub.MyClass`
  - The source file is

    *`sourcepath`*`/mypackage/sub/MyClass.java`
    - *`sourcepath`* is one or more directories
  - The class file is

    *`classpath`*`/mypackage/sub/MyClass.class`
    - *`classpath`* is one or more directories or `.jar` files potentially including the *`sourcepath`*

# Organizing Classes and Packages in the File System

- The “classpath” specifies the directories for `.class` files. The classpath can be specified:
  - on command line using `-cp` option:

    ```
    java -cp bin mypackage.MyClass
    ```

    - Finds the file `bin/mypackage/MyClass.class`
    - Runs `mypackage.MyClass.main`.
  - using `CLASSPATH` environment variable
  - by specifying the classpath to your IDE (jGRASP, IntelliJ, …)

- On the command line, in the absence of a `CLASSPATH` environment variable or a `–cp` option, the default is the current directory (`-cp .`).

# Organizing Classes and Packages in the File System

- You can have multiple directories on a classpath. The directories are separated by ‘:’ (not Windows) or ‘;’ (Windows). For example (Windows):

```
java –cp .;bin mypackage.MyClass
```

  - Directories are searched in the order given in the classpath.

- A `.jar` file can be used in a classpath as if it were a directory. For example (not Windows):

```
java –cp bin:p.jar mypackage.MyClass
```

# Compilation `javac` Command

- The Java compiler requires both the sourcepath and a location for generated class files
  - By default generated class files are co-located with the corresponding source file
  - Compiler needs to know what other classes are in the same package as the class being compiled and where to look for imported classes

```
javac –sourcepath src –d bin src/*.java
```

- Compiles all java files in `src`, but not subdirectories, placing the generated class files in `bin`. If `–d` is omitted, the default is the directory containing the source file

# Compilation
# javac Command

```
javac –cp bin –sourcepath src –d bin
src/mypackage/MyMain.java
```

- The compiler will search for imported/referenced classes in the class path and in the sourcepath.
- By default:
  - If both a class file and a source file are found, the newer will be used.
  - If a source file is found, it will be compiled and the resulting .class saved as indicated.
- The above command will resulting in compiling all source files that are (1) referenced either directly or indirectly from MyMain, and (2) where there is no .class in bin or the .class in bin is older than the source in src.

# Packages and `import`
# Sub-Packages (not!)

- Naming of packages would lead one to believe that `mypackage.sub` is, in some way, inside `mypackage`. It isn't! They are two completely separate packages.
- In particular:
  - `import java.util.*;` does not do or imply anything about the package `java.util.jar`.
  - Classes in `java.util.jar` have no access to anything non-public in `java.util`.

# Outline

- Java Background
- Java Style
- Structuring a Java program
  - Packages and `import`
  - Accessibility (`public`, `private`, etc.)
  - Style Rules
- Miscellaneous topics

# Accessibility

- Top-level types and all members (attributes, methods, inner classes) can be marked for accessibility.

- There are four levels of accessibility (most to least restrictive):
  - `private`—Can only be accessed by code in the <u>same top-level </u>type.
  - No access specified—Can be accessed by code within the same package. This is referred to as *default* or *package-private* accessibility.
  - `protected`—Can be accessed by subclasses of the given class <u>and</u> code within the same package.
  - `public`—Can be accessed anywhere.

- Top-level types can only have `public` or default access.
- Accessibility for members requires that the enclosing class have the same or a less restrictive accessibility.

# Accessibility
# Interesting Examples

```
package mypackage;
class MyClass {
    public int x;
}
```

- `MyClass` and `x` are package-private (only accessible by other classes in `mypackage`.)
  - The public accessibility of `x` is limited by the package-private accessibility `MyClass`.

# Accessibility
# Interesting Examples

```
class MyClass {
    static class SubClass {
        private static int x;
    }

    void f() {
        // x is accessible here.
        System.out.print(SubClass.x);
    }
}
```

- `MyClass` and `MyClass.SubClass` both have default or package-private accessibility.
- `x` is accessible in `f` because both `SubClass` and `f` are in the same top level class (`MyClass`).

# Outline

- Java Background
- Java Style
- Structuring a Java program
  - Packages and `import`
  - Accessibility (`public`, `private`, etc.)
  - Style Rules
- Miscellaneous topics

# Structuring Style Rules

- Only one top-level class, interface, or enum per source file.
- No classes in the default package with the possible exception of the class with the main method.
- Imports should be organized in some logical fashion.
  - Logical means: If I ask you, you can explain how you organized them. Time order, “I added them as I realized I needed them”, doesn’t count.
- The members of a class should be organized in some logical fashion.

# Outline

- Java Background
- Java Style
- Structuring a Java program
- Miscellaneous topics
  - Lambda Expressions
  - StringBuilder
  - Assertions
  - System.exit
  - NullPointerException

# Lambda Expressions

- This is a feature added in Java 8.
- This will be a quick intro.

# Lambda Expressions Example
# Sorting a List

- The sort method in the `List<E>` interface:

```
void sort(Comparator<? super E> c)
```

- Comparator:

```
@FunctionalInterface
public interface Comparator<T> {
    int compare(T o1, T o2)
    // Return negative if o1 < o2, etc.

    // And some other stuff--see docs }
```

# Lambda Expressions

- `@FunctionalInterface???`
  - A functional interface is one that has exactly one non-static, non-default method.
  - That is, it describes a class which provides a single method.
  - This is an annotation like @Override—not required, but recommended
- Use-case—the sort method
  - What if there is not a natural order? How do I specify the order to be used.
    - Answer: Provide a comparator that specifies the order.
  - Note: When does a type T have a "natural order"?
    - Answer: When `T implements Comparable<T>`.

# Java 7 example

```
class Point {
  int x, y; }

void doSortX(List<Point> l) {
  Comparator<Point> c =
    new Comparator<Point>() {
      int compare(Point p1, Point p2) {
        return p1.x – p2.x;
      } }();
  l.sort(c);
}
```

- c is an instance of a new “inner” class that implements Comparator<Point>.

# Java 8 example

```
class Point {
  int x, y; }

void doSortX(List<Point> l) {
  l.sort((p1, p2) -> p1.x - p2.x);
}
```

- The parameter to sort is a “lambda” expression.
  - It represents a function taking two parameters `p1` and `p2` and returning `p1.x – p2.x`.
  - All the required type and context information is inferred from the fact that sort’s parameter is an instance of a functional interface.

# Some lambda expressions

- One parameter, inferred type, expression result:

```
p -> p.getAge() >= 18
```

- Other than one parameter, inferred type(s), expression result

```
(a, b) -> a + b
() -> "Oops!"
```

- Block result, returns value

```
(p1, p2) -> {if (p1.x != p2.x)
               return p1.x – p2.x;
             else
               return p1.y – p2.y; }
```

- Block results, void return

```
() -> {System.out.println("Oops!");}
```

- Typed parameters (all must be typed or all must be inferred)

```
(float a, int b) -> a + b
```

# Lambda Expressions
# Method References

- When a method will work—satisfies a functional interface—you can use a method reference:

| Kind | Example |
|---|---|
| A static method of a type | `Person::compareByAge` |
| A method of a particular object | `provider::compareByName` |
| A method of an object of a particular type* | `String::compareToIgnoreCase` |
| A constructor of a class | `HashSet::new` |

* The actual object becomes the first parameter. So, String::compareToIgnoreCase(String a, String b) -> a.compareToIgnoreCase(b)

# Lambda Expressions
# Why?

- To support the new streams feature that was added to collections in Java 8. Here's an example—compute the average age of all males in a `roster` (a `List<Person>`):

```
double average = roster
    .stream()
    .filter(p -> p.getGender() == Person.Sex.MALE)
    .mapToInt(Person::getAge)
    .average()
    .getAsDouble();
```

# Outline

- Java Background
- Java Style
- Structuring a Java program
- Miscellaneous topics
  - – Lambda Expressions
  - – StringBuilder
  - – Assertions
  - – System.exit
  - – NullPointerException

# StringBuilder

- How do you build a large String from small pieces?
- Obvious solution:

```
String result = "";
for ( . . . ) {
    result = result + nextPiece();
}
// result is the desired String
```

# What's wrong?

- This algorithm is O($N^2$).
- Java's String is a value type (can't be changed)

  `result = result + nextPiece();`

  - copies `result` (at least) twice
    - creates a temporary work space
    - copies old version of `result` to the temporary
    - copies *`nextPiece`*`()` to the temporary
    - copies the temporary to new version of `result` (a new `String` object)

# Doing it right

- Do this:

```
StringBuilder temp = new StringBuilder();
for ( . . . ) {
    temp.append(nextPiece());
}
String result = temp.toString();
```

- This algorithm is O(N).
  - StringBuilder maintains a buffer which is as large or larger than the contents. When the buffer gets too small, the size of the buffer is doubled.
    - The temporary in the prior slide is a StringBuilder object
  - StringBuilder also allows modification of its contents.

# Outline

- Java Background
- Java Style
- Structuring a Java program
- Miscellaneous topics
  - Lambda Expressions
  - StringBuilder
  - Assertions
  - System.exit
  - NullPointerException

# Assertions

- This statement:

  ```
  assert expression : message;
  ```

  is equivalent to

  ```
  if (!expression) {
      throw new AssertionError(message);
  ```

- Use assertions as a way to check that your program is working the way you expect it to.
- Example:

  ```
  assert x != 0 : "x is zero";
  ```

- Make sure assertions are enabled (see next slide).

# Enabling Assertions

- Java does not enable assertions by default.
  - If you’re running your program from the command line use:

    ```
    java –ea ... MyMain
    ```
- If you’re running it using an IDE, check the documentation for the IDE. For example,
  - In jGrasp, it’s an option on the Build menu.
  - In IntelliJ and Eclipse specify `-ea` as part of the VM parameters in the Run Configuration dialog.
- Good practice: Enable assertions.
  - I will run your programs with assertions enabled.

# Assertions
# No Side Effects

- In this statement:

  ```
  assert expression : message;
  ```

  the evaluation of `expression` and `message` should have no side effects. This means
  - No changes to any variable/attribute values
  - No I/O
  - No method/function calls that do any of the above
- Why not? If there are side effects, your program may behave differently if assertions are enabled/disabled.

# Outline

- Java Background
- Java Style
- Structuring a Java program
- Miscellaneous topics
  - Lambda Expressions
  - StringBuilder
  - Assertions
  - ➡ System.exit
  - NullPointerException

# `System.exit()` No!

- Don’t call `System.exit`.
  - `System.exit(n)` causes the program to exit with return code of n.
- Why?
  - This messes up my test programs.
- What do I do if I’m in main()?
  - Generally, just return from main or see next answer.
- What do I do if I’m buried 15 layers deep in my program and I can’t proceed due to …?
  - Throw something that won’t be caught. All of Exception, RuntimeException, IllegalArgumentException, IllegalStateException, or AssertionError work.

# Outline

- Java Background
- Java Style
- Structuring a Java program
- Miscellaneous topics
  - Lambda Expressions
  - StringBuilder
  - Assertions
  - System.exit
  - NullPointerException

# NullPointerException Example

- Program (BT.java)

```
class BT {
  static void f(Object x) {
    System.out.println(x.toString());
  }

  public static void main(String[] args) {
    Object x = null;
    f(x);
  }
}
```

- Result: Backtrace

```
Exception in thread "main" java.lang.NullPointerException
        at BT.f(BT.java:3)
        at BT.main(BT.java:8)
```

Exception occurred in method BT.f at line 3 of BT.java.

BT.f was called from BT.main at line 8 of BT.java.

# NullPointerExceptions

- Suppose your program fails with a NullPointerException. The backtrace sends you to a line like:

```
x.y = a.f(b).c;
```

- How many opportunities are there for a NullPointerException?

# NullPointerExceptions

- Answer, in order:
  1. a is null—the  method a.f cannot be resolved
  2. The call a.f(b) throws a NullPointerException
     - If this happens the backtrace should point you into f.
  3. a.f(b) returns null—the attribute reference a.f(b).c cannot be resolved
  4. x is null—the assignment to x.y cannot occur
  5. If x.y is an int (or other primitive type) and a.f(b).c is an Integer (or other corresponding reference type) and is null then the attempt to “unbox” the value results in a NullPointerException.
     - The unboxing is actually a.f(b).c.intValue()
- In general, every dot “.” is an opportunity for a NullPointerException.
- One additional case: x[i] is a NullPointerException if x is null.

# References

StringBuilder

- Java StringBuilder tutorial, http://docs.oracle.com/javase/tutorial/java/data/buffers.html
- Java StringBuilder class documentation, http://docs.oracle.com/javase/8/docs/api/java/lang/StringBuilder.html

Enumerations

- Java tutorial on enums, http://docs.oracle.com/javase/tutorial/java/javaOO/enum.html
- Java Language Spec on enums, http://docs.oracle.com/javase/specs/jls/se8/html/jls-8.html#jls-8.9
- Java Enum class documentation, http://docs.oracle.com/javase/8/docs/api/java/lang/Enum.html
- *Java in a Nutshell* on enums, section 4.2, http://proquestcombo.safaribooksonline.com/book/programming/java/9781449371296/enums-and-annotations/javanut6_chp_4_sect_3_1_html

# References

Packages and import

- Java tutorial on packages, http://docs.oracle.com/javase/tutorial/java/package/index.html
- Java Language Spec on packages, http://docs.oracle.com/javase/specs/jls/se8/html/jls-7.html

Accessibility

- Java tutorial on accessibility, http://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html
- *Java in a Nutshell* on accessibility, section 3.5, http://proquestcombo.safaribooksonline.com/book/programming/java/9781449371296/data-hiding-and-encapsulation/javanut6_chp_3_sect_5_1_html

# References

Generics

- Java tutorial trail on generics, http://docs.oracle.com/javase/tutorial/java/generics/index.html

Collections

- Java tutorial trail on collections, http://docs.oracle.com/javase/tutorial/collections/index.html
- Java Library description for java.util, http://docs.oracle.com/javase/8/docs/api/java/util/package-summary.html
  - See this for documentation of Collection, List, Set, Deque, ArrayList, ArrayDeque, LinkedList, HashSet, HashMap, Iterator, ListIterator

Lambdas

- Java tutorial on lambdas (part of tutorial on nested classes) http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html
- Java tutorial on aggregate operations (part of tutorial on Collections) http://docs.oracle.com/javase/tutorial/collections/streams/
- Java Language Specification on lambdas, http://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html#jls-15.27

# References

Garbage Collection

- *GC FAQ*, http://www.iecc.com/gclist/GC-faq.html
- Wikipedia: *Garbage collection (computer science)*, http://en.wikipedia.org/wiki/Garbage_collection_(computer_science)
- c2.com: *Garbage collection*, http://c2.com/cgi/wiki?GarbageCollection
- David Detlefs, Al Dosser, Benjamin Zorn, *Memory Allocation Costs in Large C and C++ Programs*, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.30.3073
- Matthew Hertz, Emery Berger, *Quantifying the Performance of Garbage Collection vs. Explicit Memory Management*, ACM SIGPLAN Notices, Volume 40, Issue 10 (October 2005), http://portal.acm.org/citation.cfm?id=1103845.1094836
- Ed Lycklama, *Does Java Technology Have Memory Leaks*, Presentation given at JavaOne conference, 1999
- Brian Goetz, *Java theory and practice: Urban performance legends,* http://www.ibm.com/developerworks/java/library/j-jtp04223.html
- Brian Goetz, *Java theory and practice: Urban performance legends, revisited*, http://www.ibm.com/developerworks/java/library/j-jtp09275.html

# References

Garbage Collection (part 2)

- *If Only We'd Used ANTS Profiler Earlier...*
  http://www.codeproject.com/KB/showcase/IfOnlyWedUsedANTSProfiler.aspx
  - This article discusses the problem with Princeton's Grand Challenge entry.
- *A Garbage Collector for C and C++*
  http://www.hpl.hp.com/personal/Hans_Boehm/gc/
  - This is the web site for the BDW garbage collector.
- *Real-time Java, Part 4: Real-time garbage collection*
  http://www.ibm.com/developerworks/java/library/j-rtj4/index.html
- *Garbage Collectors – Serial vs. Parallel vs. CMS vs. G1 (and what's new in Java 8)*
  http://blog.takipi.com/garbage-collectors-serial-vs-parallel-vs-cms-vs-the-g1-and-whats-new-in-java-8/

Style

- Oracle Java Coding Conventions, http://www.oracle.com/technetwork/java/codeconvtoc-136057.html
  - This document is badly out-of-date but still has a lot of valuable information.
- Google Java Style Guide, https://google.github.io/styleguide/javaguide.html