Introduction to Java Programming

Barry Feigenbaum, Ph.D.

Barry Feigenbaum, Ph. D.

- 30+ years professional experience
- Using Java since version 1.0.2 (1996)
- Ph. D. Computer Engineering
- Adjunct: UT Austin, St. Edwards, FAU, ACC
- Published multiple articles, presentations (ex. JavaOne)

Note: some images and code examples used in this presentation were obtained from the internet.

Presentation Scope

- Introduce Object-Oriented Programming
- Introduce Java (mostly at Java 7 level)
- Build basic Java/JRE programming competence
  - This class will not make you a Java/JRE programming expert; that takes years of experience
  - Will only touch on small part of the JRE capabilities

Prerequisites

- Basic understanding of procedural programming
  - Some basic familiarity with programming languages.
- Basic use of personal computer
- Basic knowledge of computer systems
  - CPU, Memory, Storage, Display, etc.
- Basic knowledge of mathematics (algebra)

Presentation Description

- Introduces the idea of Object Oriented Programming, objects, classes, state and behavior, statements and expressions, instance and class methods, casting, arrays, logic and loops, creating classes, creating Java applications, command-line arguments, constructor methods, overriding methods, overriding constructors, beginning Collections usage, and finalize methods. Covers inheritance, collections, enums, exception handling, auto boxing and basic IO via the keyboard and console.

- Recommended Reference: *The Java Tutorial: A Short Course on the Basics*, Sharon Zakhour, Sowmya Kannan, Raymond Gallardo, 6th edition, ISBN 978-0134034089.

Overview of Topics

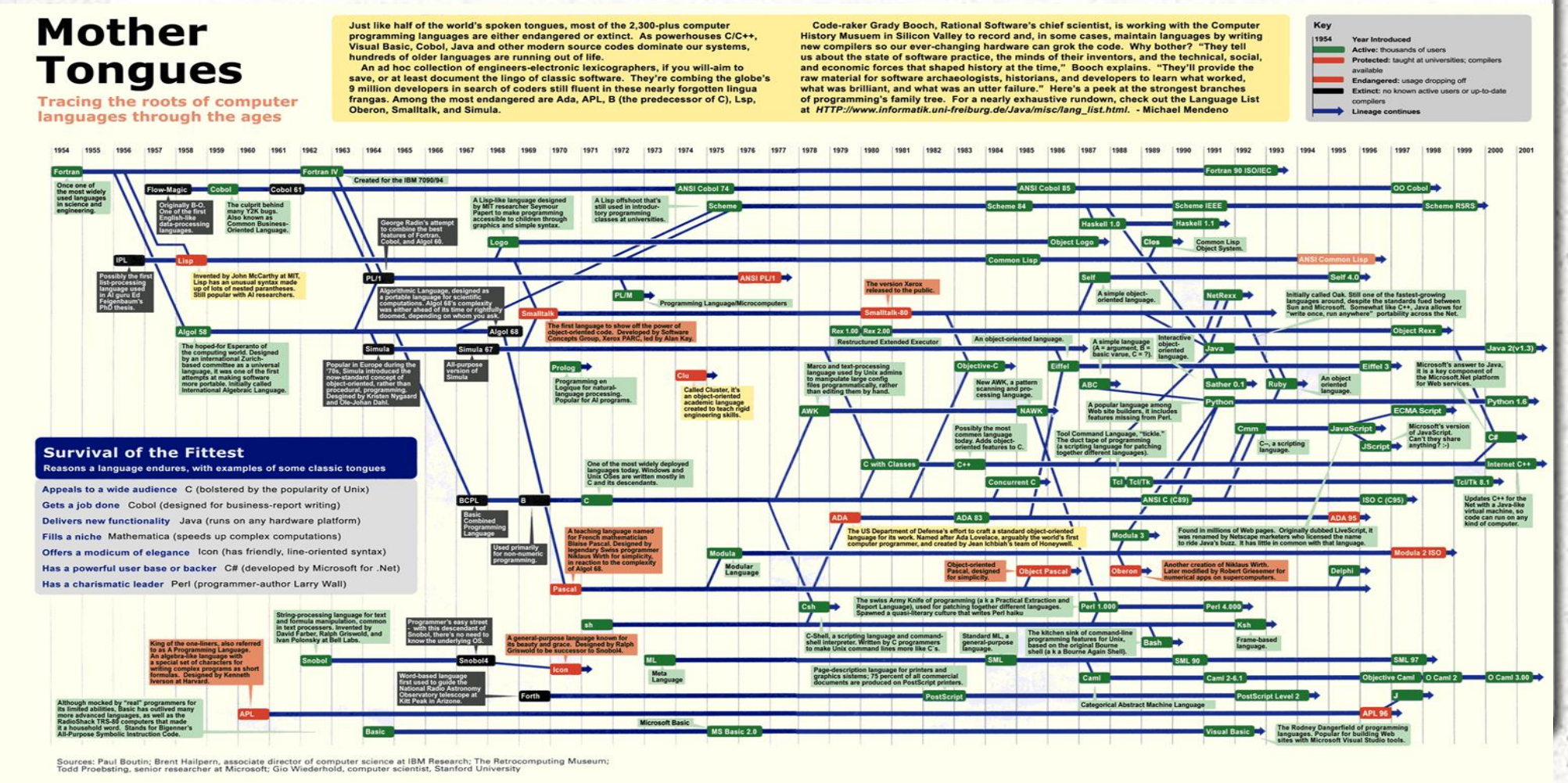
- Problem Modeling (OOA/OOD)
  - Class-Responsibility-Collaboration (CRC)
  - Unified Modeling Language (UML)
- Object-Oriented Programming (OOP)
  - State, behavior, identity, type
  - Classes, inheritance, encapsulation, polymorphism, overriding, overloading

Overview of Topics (cont)

- Java Types (classes and interfaces)
  - Fields, methods, constructors, nested types
- Java Programs
  - Packages, imports, main()
- Java Statements, Operators and Expressions

History of Programming Languages

- Long and varied history



Key Language Categories

- Machine/Assembler (HW oriented)
- Goto-rich (Spaghetti) – Fortran, COBOL
- Structured/Goto-free – C, Algol, Pascal, PL/I
- Modular – Modula, Ada
- Object-Oriented – Smalltalk, C++, Java, Python
- Scripting – Batch/Shell, Perl, Python
- Functional – Lisp, Clojure, Haskell
- Special Purpose – SQL, JCL, APL, RPG

OOx

- OOA – Object-Oriented Analysis
  - What needs to be done (C-R-C, UML)
  - High-level, close to stakeholders
- OOD – Object-Oriented Design
  - How to make it happen (UML)
  - Mid to Low-level, more details
- OOP – Object-Oriented Programming
  - Low-level, actual implementation

OOx in Brief

- Model *programs* as sets of cooperating *objects* that send *messages* to each other to make progress
- Objects are described by *classes*
- Classes can *inherit* (extend) from other classes
- Behavior can be *polymorphic* (AKA virtual)
- State can be *encapsulated*

Objects in Brief

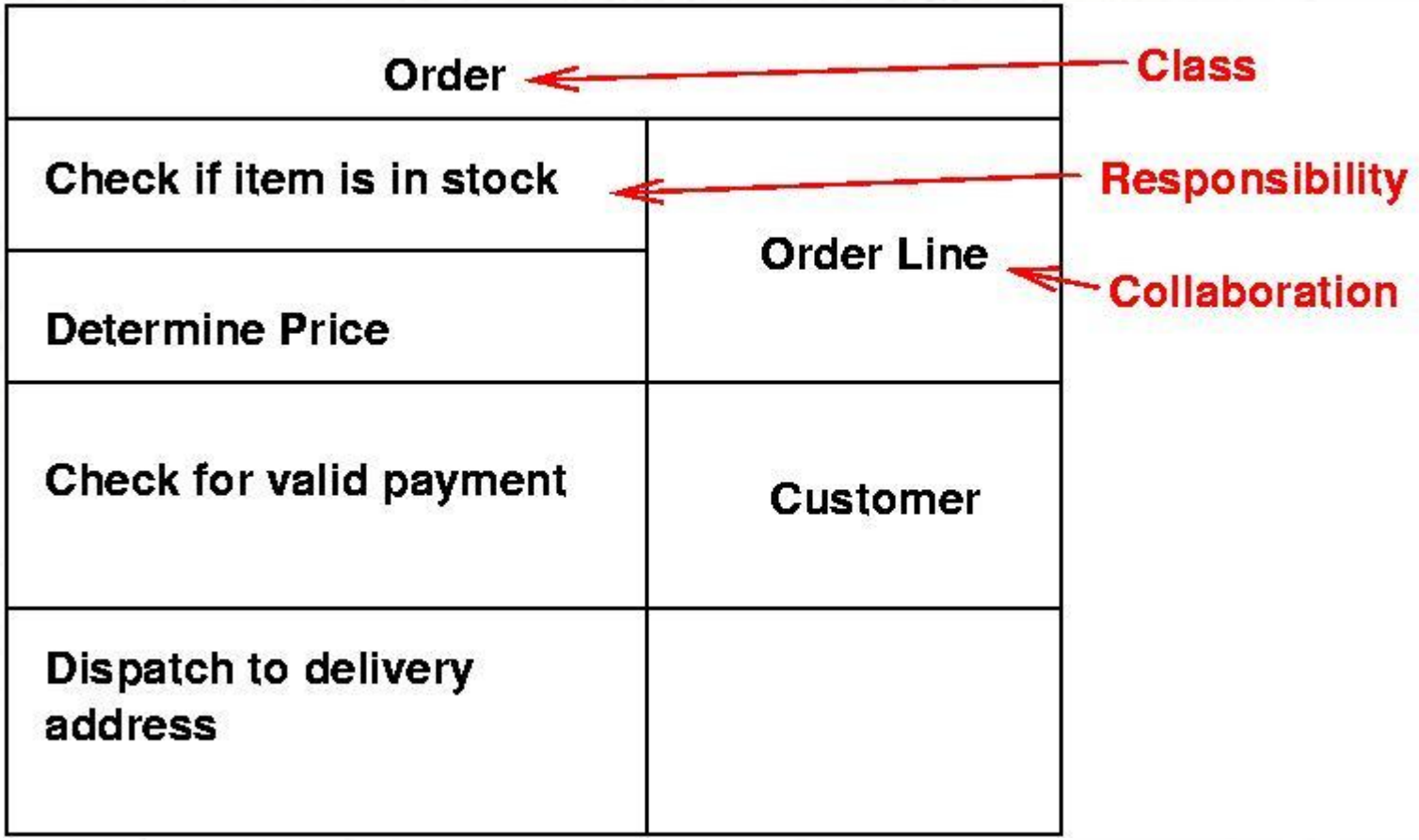
- An O*bject* is the basic collection of information in an OOP program
- Objects have key characteristics:
  - Identity (can tell one object from another)
  - State (AKA properties, **fields**, instance variables, member variables)
  - Behavior (AKA **methods**, member functions)
  - Type (generally the class that created it)
  - Lifetime – how long they exist

OOA via C-R-C

- *Class-Responsibility-Collaboration* (CRC) cards are a tool used in the design of object-oriented software.
- Created from index cards with three areas:
  - Top: the class name
  - Left: the responsibilities of the class
  - Right: collaborators (other classes) that interact with this class

Example C-R-C Card

- Constrained to size of index card



Creating C-R-C Cards

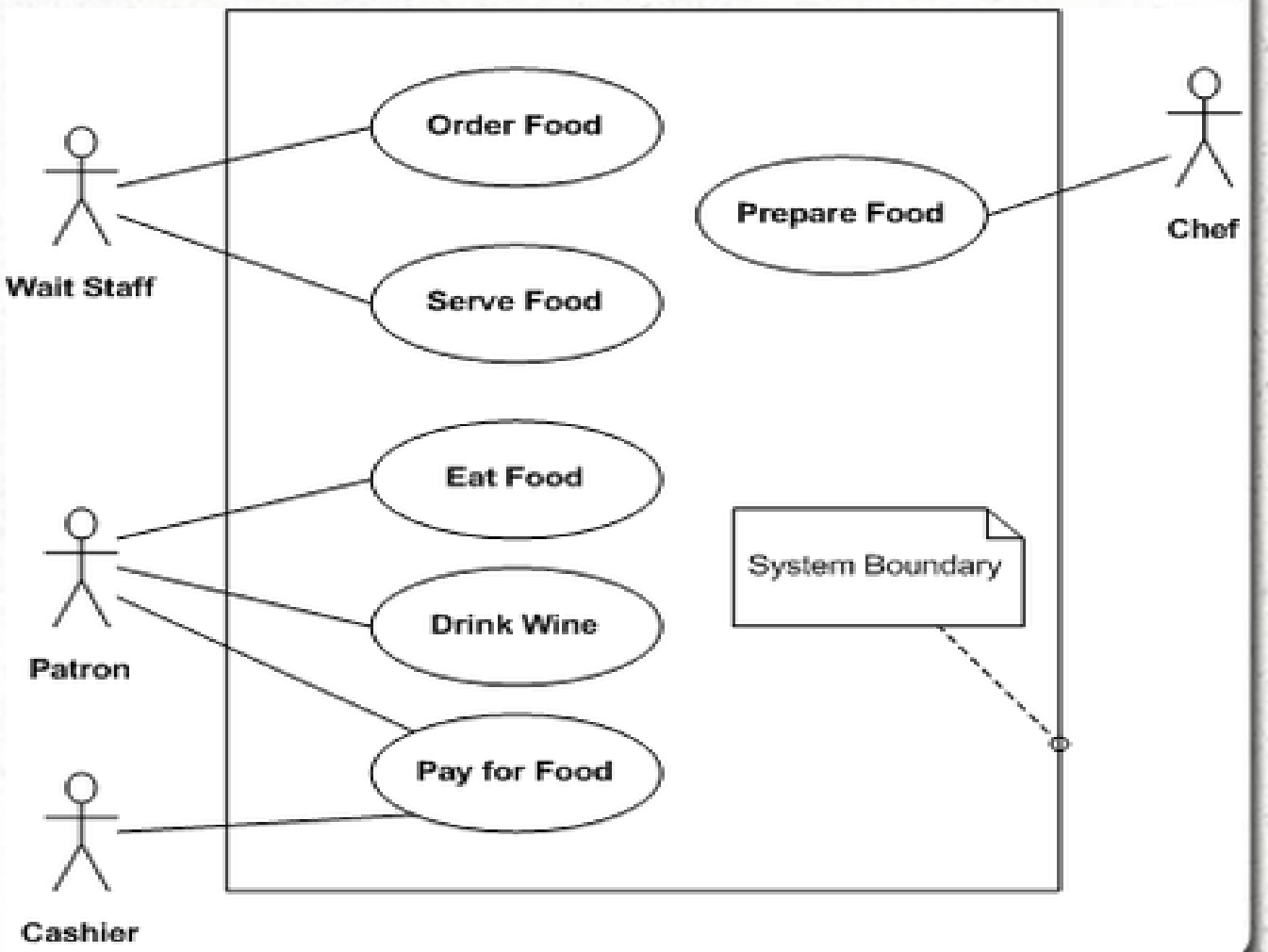
- Create a scenario identifying the major actors and actions of the actors; only include actions and actors specific to the problem
  - Nouns become classes or properties
  - Verbs become the responsibilities
  - Collaborators are other cards this card interacts with

OOA & OOD via UML

- *Unified Modeling (visual) Language*
  - *Use Case Diagram* - static
  - *Class Diagram* - static
  - *{Object} Interaction Diagram* - dynamic
  - *State {Transition} Diagram* - dynamic
  - Many more – will not not discuss
  - See http://creately.com/blog/diagrams/uml-diagram-types-examples/

Use Case

- Describe interactions from a User's point of view

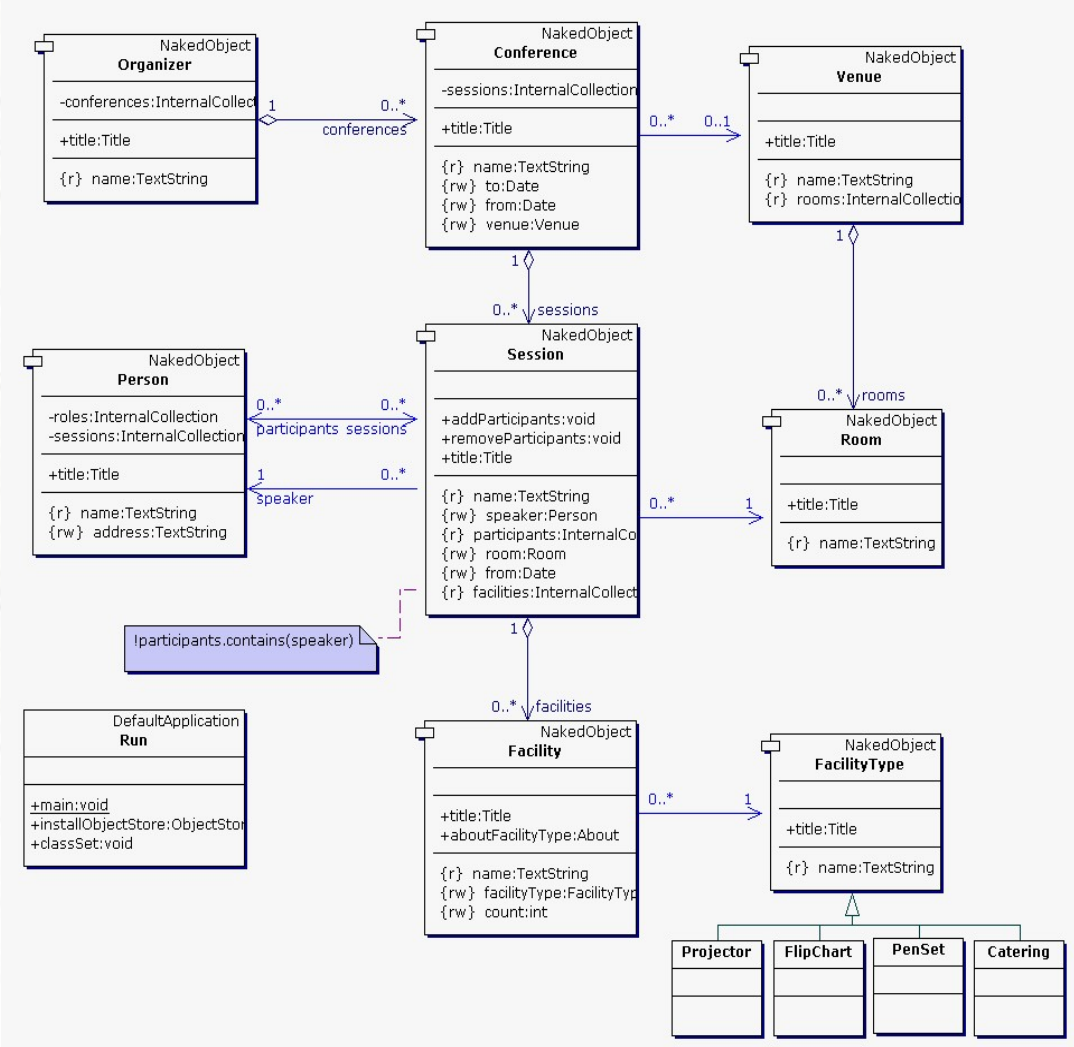


Use Case Elements

- Title – brief but unique
- Description – sufficient to understand
- Happy Path Steps – process that works
- 0+ Unhappy Paths Steps – alternate paths
- Key elements
  - Actors – who/what participates
  - Operations – what happens
  - Flow – in/between use cases

Class Diagram

- Types
  - Interfaces
  - Properties
  - Methods
- Relationships
  - Reference
  - Inheritance



Inheritance Relationship

- An “IS-A” relationship
  - Generalized – Superclass – Parent
  - Specialized – Subclass – Child
- Only use when IS-A (ex. cat is-a animal) pertains
  - Do no use to reuse implementation
- Single Inheritance – Java Class
- Multiple Inheritance – Java Interface

Other Relationships

- General (or Simple)
  - Members cooperate
- Composition (sometimes Delegation)
  - Has-A – component lifetime independent of container
- Containment
  - Part-Of – component lifetime tied to container

Relationship Cardinality

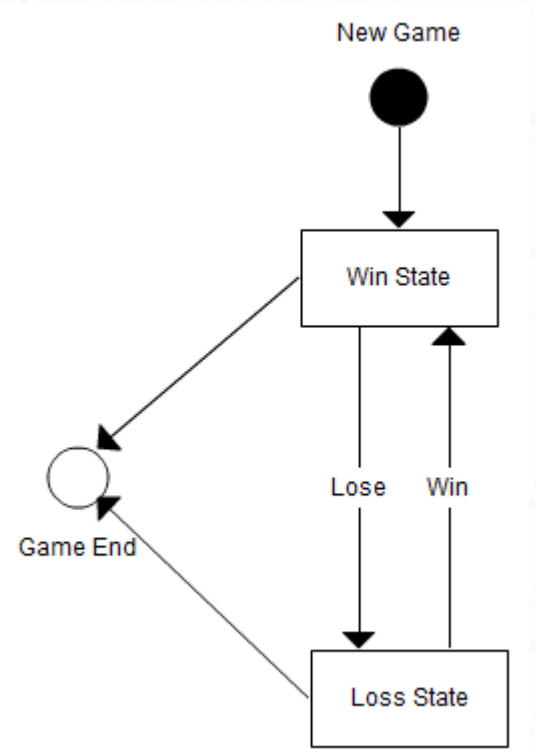
- Each end of relationship has cardinality
- 0..1 – optional (often entered as ?)
- 1..1 – required
- 0..n – multiple (often entered as *)
- 1..n – multiple (often entered as +)
- Common: 0..1, 1..1, 1..m, m..1

Inheritance vs Relationship

- Use Inheritance only to say “IS-A”
  - Never to reuse implementation
  - Use composition/delegation instead
- Use Relationships to define collaborations
  - Cooperation
  - Part-of

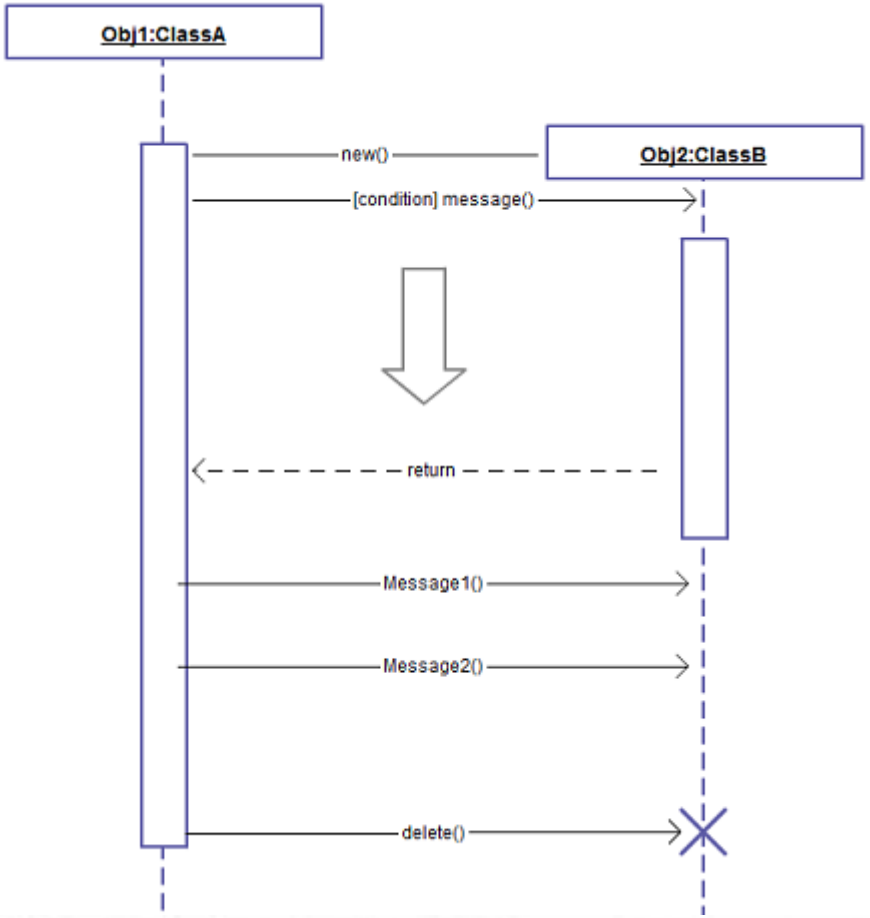
State Transition Diagram

- Start/Stop States
- Other states
- Valid transitions and events that cause the transition



Interaction Diagram

- Shows sequence of messages between 2+ objects over time



OOP in Java

- Java has classes, abstract classes, interfaces
- Classes have fields, methods, nested types
- Classes have instances (created by *new*)
- Classes have exactly one super-class (*extends*)
- Classes can *implement* many interfaces
- *java.lang.Object* is root super-class
- Static vs. instance fields/methods

Java Ecosystem

- Java Language
  - Generally what unqualified “Java” means
- Java Tools
  - Compiler, IDE, etc.
- Java Run time (JVM and JRE)
  - Interpreter, JIT, classes and byte-code
- Java Class Libraries
  - Standard and third party (very large set)

Java Program Life-cycle

- Edit/compile Java source(s)
  - edit Xxx.java
  - javac Xxx.java
- Xxx.java → Xxx.class (and maybe others)
- Launch program: java Xxx
  - Starts at:

```
public static void Main(String[] args)
```

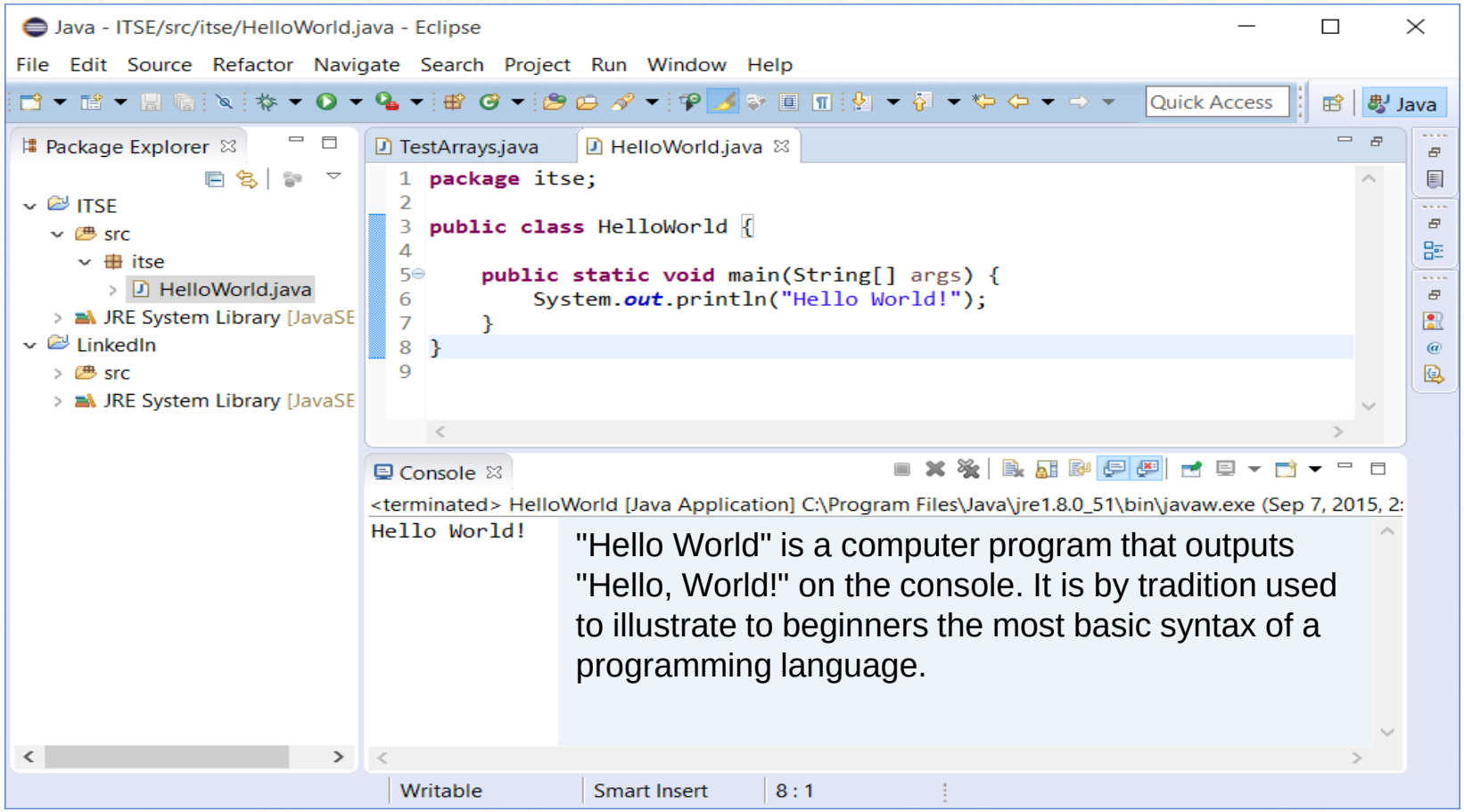
- Interact with program (optional); Program exits

Java Binding

- Each public Java class can be a program
  - A program is a sea of classes
- All classes are loaded dynamically (on demand)
  - All linking between classes is dynamic
  - Classes can reside locally or remotely
- Using ClassLoaders and classpaths
  - Very different from static c/C++
  - Somewhat like Shared Objects or DLLs

IDE - Eclipse

- Edit/-compile-run all in IDE environment



Key Java Characteristics

- "C++" style syntax and statements
- *Managed* objects with *garbage collector*
- Statically typed – run-time type checking
- Dynamically loaded and linked
- Network enabled – Write Once-Run Everywhere
  - Interpreter + JIT/hotspot + classpath
- Multi-threaded; concurrent capabilities
- Object-oriented; Application-oriented

Java Editions

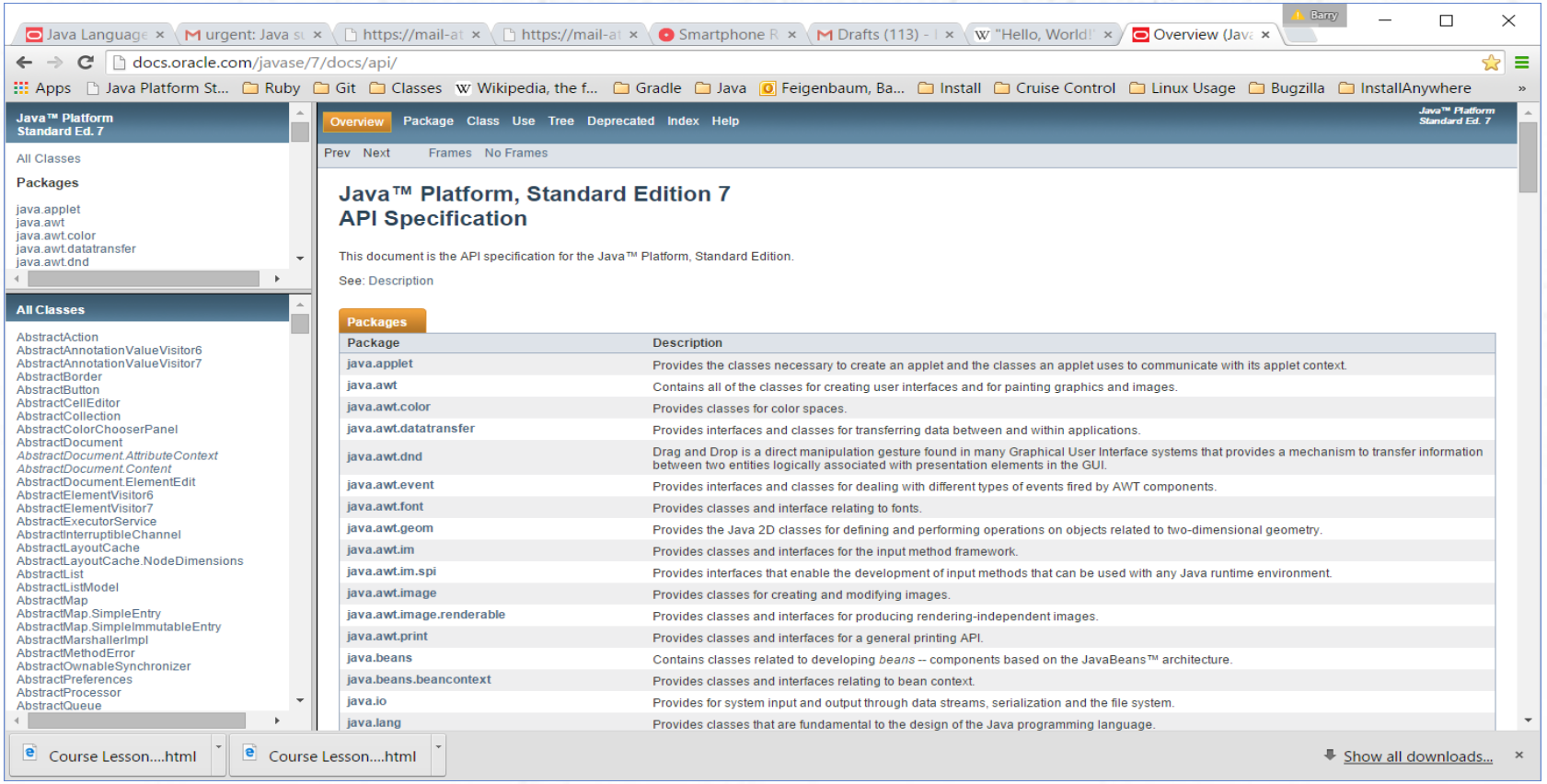
- **Standard** – JSE (used in this presentation)
  - Desktop usage
- **Enterprise** – JEE (superset of JSE)
  - Server usage
- **Micro** – JME (subset of JSE)
  - Phones and similar
- **Mobile** – on Android (parts of JSE/JEE)
  - Phones, tablets and similar

Java Runtime Size

- JSE - https://en.wikipedia.org/wiki/Java_version_history
  - Over 200 packages
  - 4000+ public types across packages
  - 10000s of public methods across classes
  - We will touch only on small % of these
- JEE much larger
- Third party libraries add millions of types

Java API

- API JavaDoc is your key reference



Java Primitive Data Types

- Byte(byte) (8-bit signed): (byte)-1, (byte)0
- Short(short) (16-bit signed): -1S, 0S, (short)-1
- Character(char) (16-bit unsigned): 'a', '1', (char)0
- Integer(int) (32-bit signed): -1, 0
- Long(long) (64-bit signed): -1L, 0L,
- Float(float) (32-bit signed): -1.0F, 0.0F
- Double(double) (64-bit signed): -1.0, 0.0
- Boolean(boolean) (unspecifed): true, false
- Void(void): <no value> (only method type)

All primitive (non-object) types have a *wrapper* (object that holds the value) type

Basic Console I/O

- Console output:
  - `System.out.print{ln}(<value>)`
  - `System.out.printf( "<pattern>", <values>...)`
- Console input:
  - `Scanner sc = new Scanner(System.in);`
  - `int i = sc.nextInt();`
  - `double d = sc.nextDouble();`
  - `String line = sc.nextLine();`

Common printf %x Formats

Conversion	Description
'b', 'B'	boolean
's', 'S'	string
'c', 'C'	character
'd'	integer
'x', 'X'	hex integer
'e', 'E'	scientific floating point
'f'	fixed floating point
'g', 'G'	general (auto scientific or fixed) floating point
't', 'T'	date/time
'%'	literal percent
'n'	literal line separator

Input (String) Parsing

- Each wrapper type has a (static) "parse<type>" method that converts a string to its type
  - `int x = Integer.parseInt("12345");`
  - `double d = Double.parseDouble("1.2e3");`
  - `boolean b = Boolean.parseBoolean("false");`

Java Reference Data Types

- Object and all sub-classes: (Heap-based: all created by **new** operator)
  - Key types:
    - java.lang.String/StringBuilder
    - Primitive Wrappers
    - All arrays

Java Reference Data Types (cont)

- java.lang.System/Runtime/Math
- Various java.util collections
  - From Java Runtime Environment (JRE)
  - From third party libraries

Object Methods

- All objects have these methods
  - See JavaDoc for all methods
- equals(Object other) – test equality
- hashCode() -gets hash for Maps (<=>equals)
- toString() - gets a string representation
- getClass() - gets the class of this instance
- clone() - makes a copy of this instance

Core Reference Types

- Key data types:
  - Java.lang.String/StringBuilder
  - Java.util.{Array}List
  - Java.util.{Hash}Map/TreeMap
  - Comparable<T>
- Key objects:
  - System.in/out/err
  - Runtime.getRuntime()

Creating Reference Types

- Use the *new* operator and a constructor call
  - Date d = new Date();
  - File f = new File(“myFile.txt”);
- String special case; can use literals
  - String s = “hello”;
- Wrappers special case; can use literals
  - Integer i = 1;

Null

- All variables of reference type can be assigned the **null** value
  - This indicates that there is nothing assigned to the value
  - This the default value for reference types
- Date d1, d2 = null; (both null)
- Test with if:
  - if(d1 == null) … or if(d1 != null) ...

Reference Aliases

- An object may have 0+ references
  - If 0, then subject to garbage collection
  - Test for with sameness (identical)
- Date d1 = …, d2 = …;
- if(d1 == d2) … or if(d1 != d2) …
  - Sameness not same as Equality
- == not same as equals()
- if(d1.equals(d2)) … or if(!d1.equals(d2)) …

Creating Collections

- Strings (list of characters)
  - String s = new String(“xxx”); (or just “xxx”)
- Lists (lists of anything)
  - List l = new ArrayList(); l.add(1); ...
  - List l = Arrays.asList(1, true, 3.5, “4”);

Creating Collections (cont)

- Maps (set of key:value pairs)
  - Map opposites = new HashMap();
  - opposites.put(“hello”, “goodbye”);
  - opposites.put(“goodbye”, “hello”);

Java Arrays

- 0+ elements of **same** type (primitive or reference – including other arrays)
- All arrays are reference (heap) objects
- Declaration: int[] intArray;
- Creation: always a fixed length
  - intArray = new int[50];
  - intArray = new int[] { 1, 2, 3, 4};
- Reference: intArray[5] = 10;
  - Get exception if index out of bounds

Key Operators

- Arithmetic: `+`[1] `-`[1] `*` `/` `%` `++` `--`
- Boolean: `&` `|` `~`
- Logical: `&&` `||` `!`
- Relational: `==` `!=` `<` `>` `<=` `>=`
- Ternary: `<cond> ? <true> : <false>`
- Assignment: `=`
- Augmented assignment: `+=` `-=` `*=` `/=` `%=` `&=` `|=`

[1] unary and binary

Key Operators

- Cast: `(int)1.45e3;`
- New: `new MyClass()`
- Method call: `xxx(1, 3, 5)`
- Dereference:

`object.method(); object.field;`

- Type test: `x instanceof MyType`

Variables

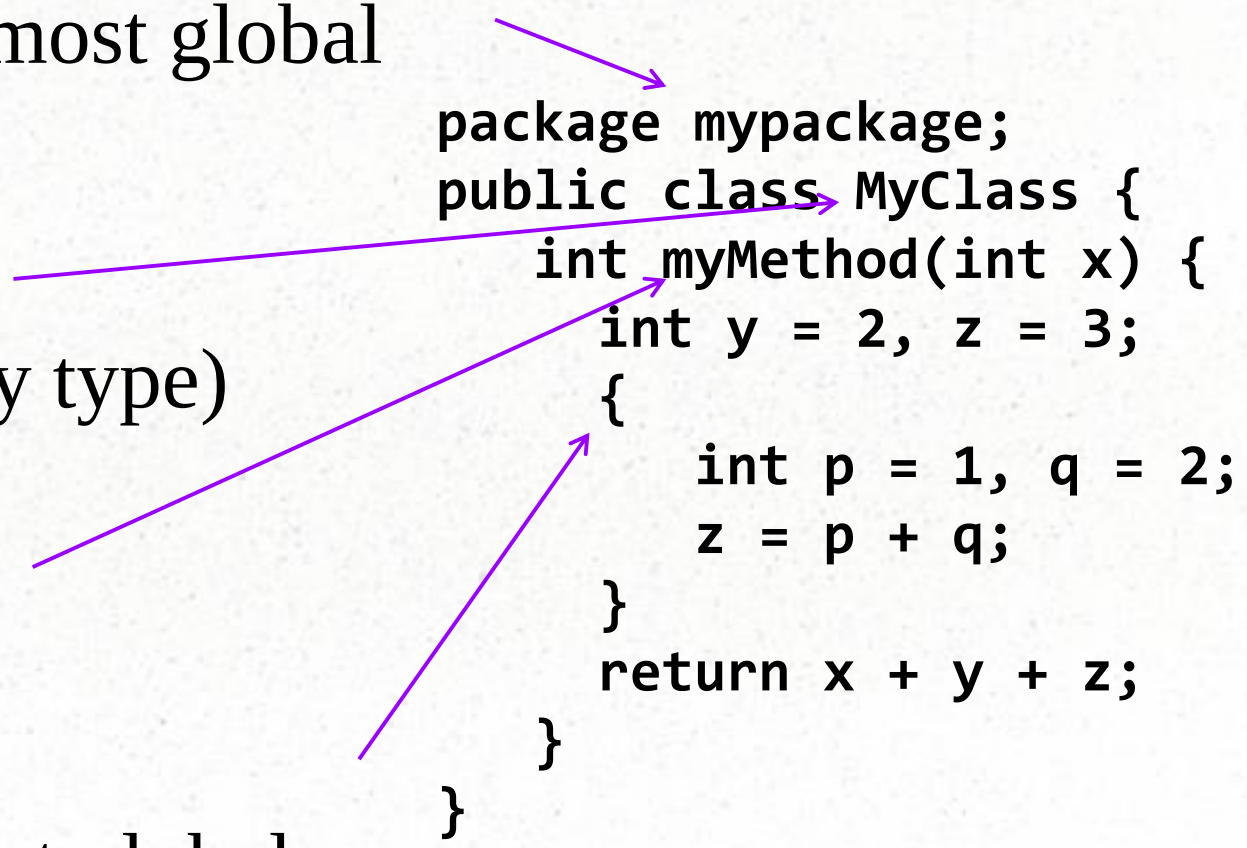
- Named reference to a storage location
  - Names are letters, digits (not first), _
  - Can hold primitive value or reference to Objects
  - Object references can be *aliases*
- Variables have *scope*
- Variables allow change over time (vs constants)

Variable Name Conventions

- Types (noun): SomeNewType
- Fields (noun): someFileName (or _someFieldName or this.someFieldName)
- Methods (verb): someMethodName
- Local names (noun): someLocalName
- Constants: SOME_CONSTANT_NAME
- Package: domain.somename.anothername

Scopes

- Package – most global
- Class
(actually any type)
- Method
- Block – least global



```
package mypackage;
public class MyClass {
    int myMethod(int x) {
        int y = 2, z = 3;
        {
            int p = 1, q = 2;
            z = p + q;
        }
        return x + y + z;
    }
}
```

Expressions

- (Legal) combination of literals, variables and operators that calculate a result value.
- Evaluation based on precedence of operators
  - Use (…) to re-order evaluation
- Examples:
  - `x = 1`
  - `x = Math.sin((1 + 2) * -3 * Math.PI)`
  - `x = “Hello” + ' ' + ”World!” + 1`

Java Reserved (key) Words

Discussed in this presentation

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

type ~~visibility~~ *statement* modifier/special

Java Source Structure
Xxx.java

- Package statement (if any)
  - Path to source must match package
- Import statements (if any)
- Types: class, interface, enum
  - Only one type can be public; it must match source file name
  - Additional private types allowed

Package Related Statements

- Package: Declares package to use
  - `package com.mycompany.myprogram;`
- Import: Add external types, methods or values
  - `import java.util.*;`
  - `import static java.lang.Math.*;`

Structure Statements

- Class: Declares a class
  - `class MyClass extends SomeClass implements SomeInterface { … }`
- Interface: Declares an interface
  - `interface MyInterface extends SomeInterface { … }`
- Method: Declares a method
  - `int calculate(int x, int y) { return x + y; }`

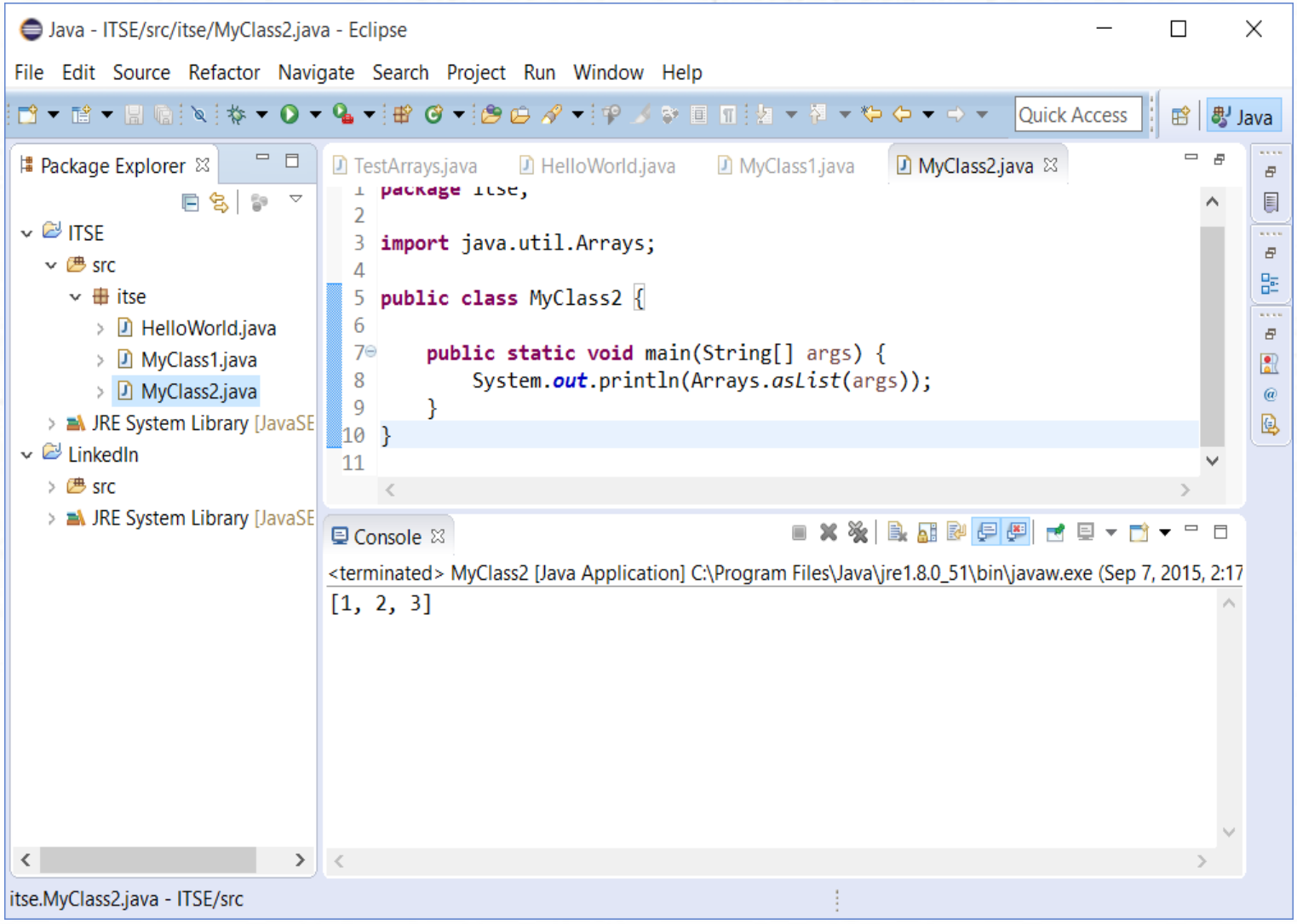
Key Method Body Statements

- Declaration: `int x = 1;`
- Assignment: `x = 1; x += 2;`
- Block: `{ … }`
- If/Else: `if(x > 1) x = 0; else x = 1;`
- While: `int i == 0;`
`while(i < 10) { …; i++; }`
- For: `for(int i =0; i < 10; i++) { … }`
- Return: `void x() { return; }`

Simple Class

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

MyClass2 in IDE



Strings vs. StringBuffer

- Strings are *immutable* (cannot change)
  - `String x = "hello";`
  - `x += "world";` (x replaced by new instance)
- StringBuilders are *mutable* (can change)

```
-StringBuilder sb = new StringBuilder();
-sb.append("hello");
-sb.append("world");
-String x = sb.toString();
```

Auto-Boxing

- *Boxing* is automatic conversion between primitive and corresponding wrapper types
- `int x = 1;`
- `Integer X = 1;  Integer Y = x;`
- `X = x;  (x is boxed)`

  -vs. `X = new Integer(x);`
- `x = X;  (X is unboxed)`

  -vs. `x = X.intValue();`

Exercise

- Create a program to create and sort content of an integer (as below) and string array

```
public static void main(String[] args) {
        int[] ia = new int[args.length];
        for (int i = 0; i < args.length; i++) {
                ia[i] = Integer.parseInt(args[i]);
        }
        System.out.println("Before: " + asList(ia));
        bubbleSort(ia);
        System.out.println("After : " + asList(ia));
}
```

- How long (vs array size) does this run?

Exercise (cont)

- Create a program to create and sort content of an integer and string array

```
static void bubbleSort(int[] num) {
        boolean anyFlipped = true;

        while (anyFlipped) {
                anyFlipped = false;
                for (int j = 0; j < num.length - 1; j++) {
                        // if out of order, swap
                        if (num[j] > num[j + 1]) {
                                int temp = num[j];
                                num[j] = num[j + 1];
                                num[j + 1] = temp;
                                anyFlipped = true;
                        }
                }
        }
```

Exercise

- Create a program to create and sort content of an integer and string array

```
private static List asList(int[] ia) {
        List l = new ArrayList(ia.length);
        for(int i = 0; i < ia.length; i++)
        {
                l.add(ia[i]);
        }
        return l;
}
```

Class Members

- Fields: represents the state
- Methods: represents the behavior
- Constructors: initialize new instances
- Nested Types: types within types

Fields

- Implements state (properties) for an object
- Can be of any type (except **void**)
- Generally should be *encapsulated*
  - Private (sometimes default/protected) field with public access (get/set) methods
  - Allows implementation to change without impacting clients

Methods

- Implements behavior associated with a type
- Contain 0+ Java statements
- Can be **void** or return a value of any type
- Can have 0+ parameters of any type (except void)
- Can be **abstract** (empty) in interface or abstract class
- Can be overloaded and overridden

Pass-by-Value

- All methods use *pass by value* parameters (and results)
  - The actual value is copied from the caller to the method as a method local
  - If the parameter is a reference type an *alias* is created
  - Changes in the local value are not seen by the caller
- Must pass reference to *holder* object to update the caller

Pass-by-Value Example

- Main

```
public static void main(String[] args) {
        int p1 = 0; int p2 = 0;
    String s1 = null;
        StringHolder sh = new StringHolder();

        System.out.printf(
        "Input values : p1=%d, p2=%d, s1=%s, sh=%s%n",
                    p1, p2, s1, sh.value);
        int result = changer(p1, p2, s1, sh);
        System.out.printf(
        "Output values: p1=%d, p2=%d, s1=%s, sh=%s,
result=%d%n",
                    p1, p2, s1, sh.value, result);
    }
```

Pass-by-Value Example

- Method called

```
    static class StringHolder {
      String value;
    }

    static int changer(
   int p1, double p2, String s1, StringHolder sh)
 {
       p1 = 1; p2 = 2;
    s1 = "s1 new value";
    sh.value = "sh new value";
         return 1;
    }
Input values : p1=0, p2=0, s1=null, sh=null
Output values: p1=0, p2=0, s1=null, sh=sh new value, result=1
```

Constructors

- Initialize instances of a class
- Named same as class name
- Constructors are like instance methods called immediately after **new**
- Contain 0+ Java statements
- Has no return type
- Can have 0+ parameters of any type (except void)
- Can be overloaded

Constructor Types

- Default: 0 arguments
  - Created automatically by compiler if no other constructors provided
  - Recommended all classes have one
- Copy: 1 argument of same type
- Full: an argument for all properties
- General: not one of above

Destructors

- Not needed, JRE does garbage collections
- Special method: protected void finalize()
  - Used to do any resource cleanup
  - Called by garbage collector before reclaiming object
  - Implemented by Object; sub-classes rarely need one

Instance Variables

- Values associated with each instance
-

```
public class Person {
  boolean male;
  String name;
  Date birthDay;

  public boolean isMale()
  { return male; }
  public void setMale(boolean male) {
this.male = male; }
}
```

Private variable + get/set method makes a *property* which *encapsulates* the state

Instance Variables

- Values associated with each instance
-
```
public class Person {
  boolean male;
  String name;
  Date birthDay;

  public String getName()
  { return name; }
  public void setName(String name) {
this.name = name; }
}
```

Instance Variables

- Values associated with each instance
-
```
public class Person {
  boolean male;
  String name;
  Date birthDay;

  public Date getBirthDate()
  { return birthDate; }
  public void setBirthDate(
    Date bd) { birthdDay = bd; }
}
```

Visibility

- **private** – can be seen only in owning class; Typical for fields
- <default> **-** can be seen in any class in the same package; avoid using this
- **protected** – can be seen in owning class or any sub-class (regardless of package)
- **public** – can be seen in any class; Typical for methods

Class vs. Instance Variables

- Class (AKA static) variables
  - Shared across all instances of the class
  - Exist with class (no instance needed)
  - Identified by **static** modifier
- Instance (AKA non-static) variables
  - Local to a single instance
- Each instance has its own copy
  - Created/destroyed with instance

Class vs. Instance Methods

- Instance (non-static) methods
  - Can access all static variables and instance (non-static) variables
  - Has an implied **this** argument
- Class (AKA static) methods
  - Can access only static variables
  - No **this** implied argument

Overloading vs Overriding

- Overloading
  - 2+ methods of same name but different signatures (number and/or types of parameters)
  - int add(int x, y)
  - List add(Object o)
- Overriding
  - Same method+signature in subclass
  - Method in parent class cannot be final

Final vs. non-final

- Final method: **final** void x() {...}
  - Cannot override
- Final field: **final** int x = 1;
  - Cannot be changed after constructor
- Final + static: **final static** int ONE = 1;
  - Effectively a named constant

Packages

- Packages group types into a name-space (the package)
  - Organized as a hierarchy
  - Named as a sequence: package x.y.z;
  - All types in some package
- If no package statement *default* package used
- Packages (also types) create name-spaces
  - Allow different types from different vendors to share same name

Imports

- Import allows unqualified names:
  - import java.lang.*;
  - Use: System (vs. typing java.lang.System)
- Static import allows static object to be used unqualified
  - import static java.lang.Math.*; (imports Math.PI)
  - Double area = 2 * PI * radius * radius;
- Always optional; often convenient

Inheritance

- A hierarchical relationship between types
  - Superclass/parent; Subclass/child
- class Mustang extends Automobile
  - Mustang has all fields and methods of Automobile; may add more
  - Can use a Mustang anywhere an Automobile is expected
- Automobile car = new Mustang();

Inheritance

- Methods can be inherited
  - Class Automobile has method start()
  - Class Mustang does not have start()
- Automobile car = new Mustang();
- car.start() - runs Automobile.start()
- The *start* method is inherited

Polymorphism

- Comes from inheritance
  - Class Automobile has method start()
  - Class Mustang has method start()
- Automobile car = new Mustang();
- car.start() - runs Mustang.start()
- The *car* variable is *polymorphic* (multi-shaped)
- Mustang.start() *overrides* Automobile.start()

Interface vs. Implementation Inheritance

- Use interfaces for interface inheritance
  - Interfaces define protocol (set of method signatures)
  - Can inherit 0+ interfaces
  - Java 8 allows “default” implementations
- Use classes for implementation inheritance
  - Classes define interface and (partial or full) implementation (method bodies)
  - Must inherit exactly 1 class (default Object)

Interface vs. Implementation Inheritance

- Each instance is an instance of its class and any classes and interfaces transitively extended
  - That is *x instanceof <anySuperType>* is true
- class MyClass extends MySuperClass implements MyInterface1, MyInterface2

Interface vs. Implementation Inheritance

- MyClass instances are *instanceof* (at least): MyClass, MySuperClass, MyInterface1, MyInterface2 and Object

```
MyClass mc = new MyClass();
MySuperClass msc = mc;
MyInterface1 mi1 = mc;
MyInterface2 mi2 = mc;
Object o = mc;
```

- No cast required but it is allowed

```
Object o = (Object)mc;
```

Upcast vs. Downcast

```
MyClass mc = new MyClass();
MySuperClass msc = mc;
MyInterface1 mi1 = mc;
MyInterface2 mi2 = mc;
Object o = mc;
:
```

```
o = (Object)mc;    (upcast - optional)
mc = (MyClass)o;   (downcast - required)
```

- Upcasts always work at run-time
- Downcasts may fail at run-time

Interface Vs. Class

- Interface defines a type (pure abstract class)
  - Has public method prototypes
  - Has public **static final** fields
  - Has default methods (Java 8+)
- Class defines a type and provides an implementation; If incomplete class must be **abstract**
  - Has instance fields and methods

Interface Usage

- Describes the behavior (methods) a type must have
  - For parameters, best to use interface
  - Prefer use interfaces as variable types:
- List myList = new ArrayList();
- Interfaces can be multiply inherited
- Classes can be only singly inherited

Java Collections

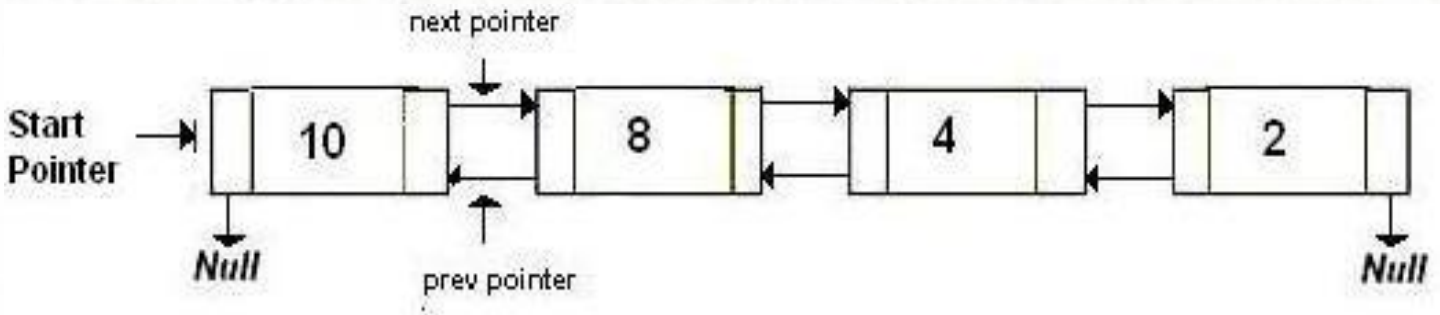
- Most in java.util package
- Key interfaces:
  - Collection: basic unordered (abstract)
  - List – ordered collection (extendable array)
  - Map - set of key/value pairs
  - Set – unordered unique values

Java Collections (cont)

- Key implementations:
  - Collection: AbstractCollection
  - List – ArrayList, LinkedList, Vector, Stack
  - Map – HashMap, TreeMap, LinkedHashMap
  - Set – HashSet, SortedSet

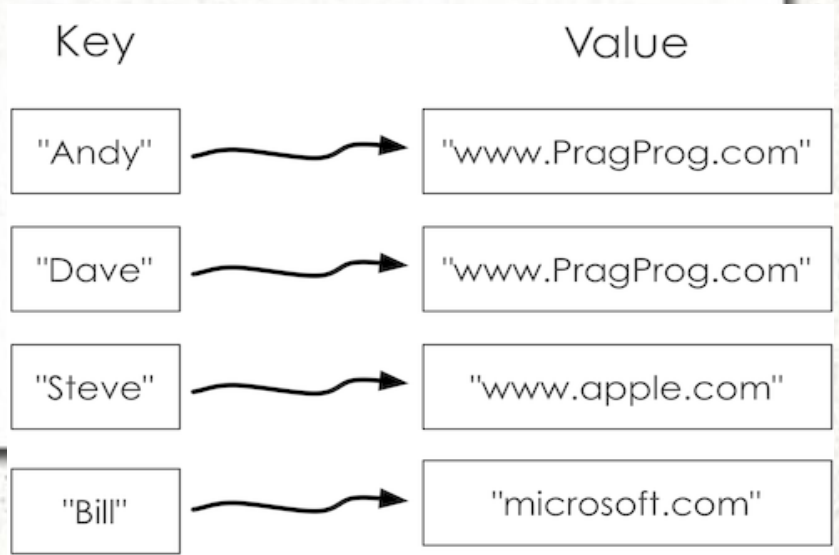
List

- Example Linked List
  - 4 elements: [10, 8, 4, 2]
- Access: O(n); Average: n/2



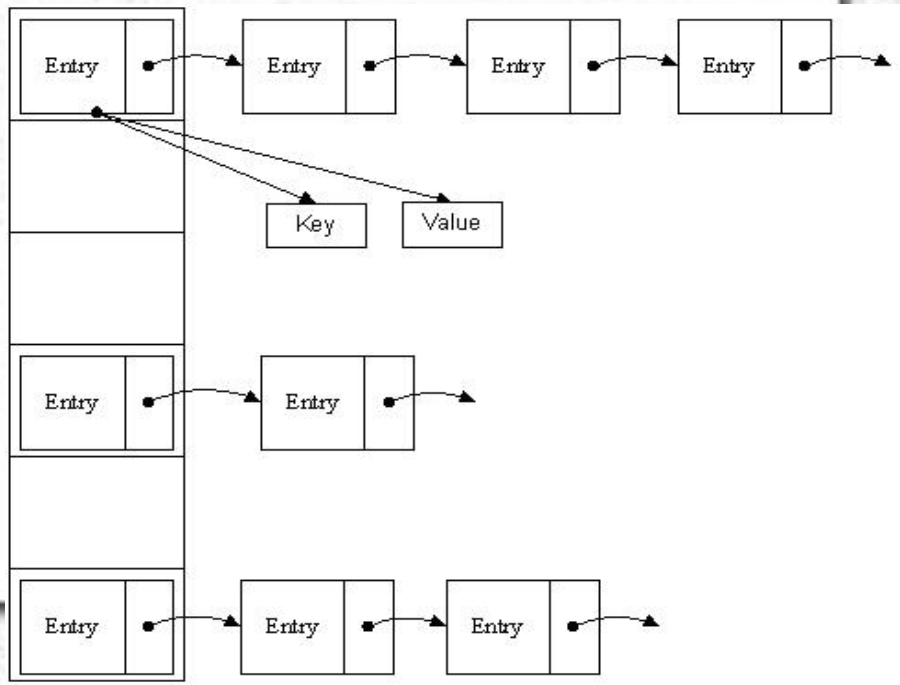
Map

- Example Map String to String
  - No particular order; key unique
  - Equals and hashCode of key must be consistent
  - Use primitive wrappers or String



HashMap

- Example Map Key to Value
  - No particular order; key unique
  - Common “bucket” implementation
- Access: O(1);
Average: 1 - 3
when well balanced and sufficient buckets
- Hash of key to select bucket



HashSet

- Example Map Key
  - No particular order; key unique
  - Common “bucket” implementation
- Access: O(1)

- Like HashMap but with no value

Hetro vs. Homo-geneous Collections

- Java collections are by default *Heterogeneous* (can hold any type)
  - List al = new ArrayList();
  - al.add(1);
  - al.add(2.5);
  - al.add("Hello");
  - Can cause run-time type exceptions

Hetro vs. Homo-geneous Collections

- Java collections can become *Homogeneous* via "Generics"
  - List<String> al = new ArrayList<String>();
  - al.add("Hello"); (allowed)
  - al.add(1); (compiler error)

Generics in Brief

- Many types, especially collections, can be "generic"
- Ex.

Map<String, Date> m = new HashMap<>()

  - m's key must be String
  - m's value must be Date

- Helps make using collections more type safe
- Hard to author; fairly easy to use

Switch Statement

- Like a compound IF/ELSE
- switch(value) { (value: int, enum or String)

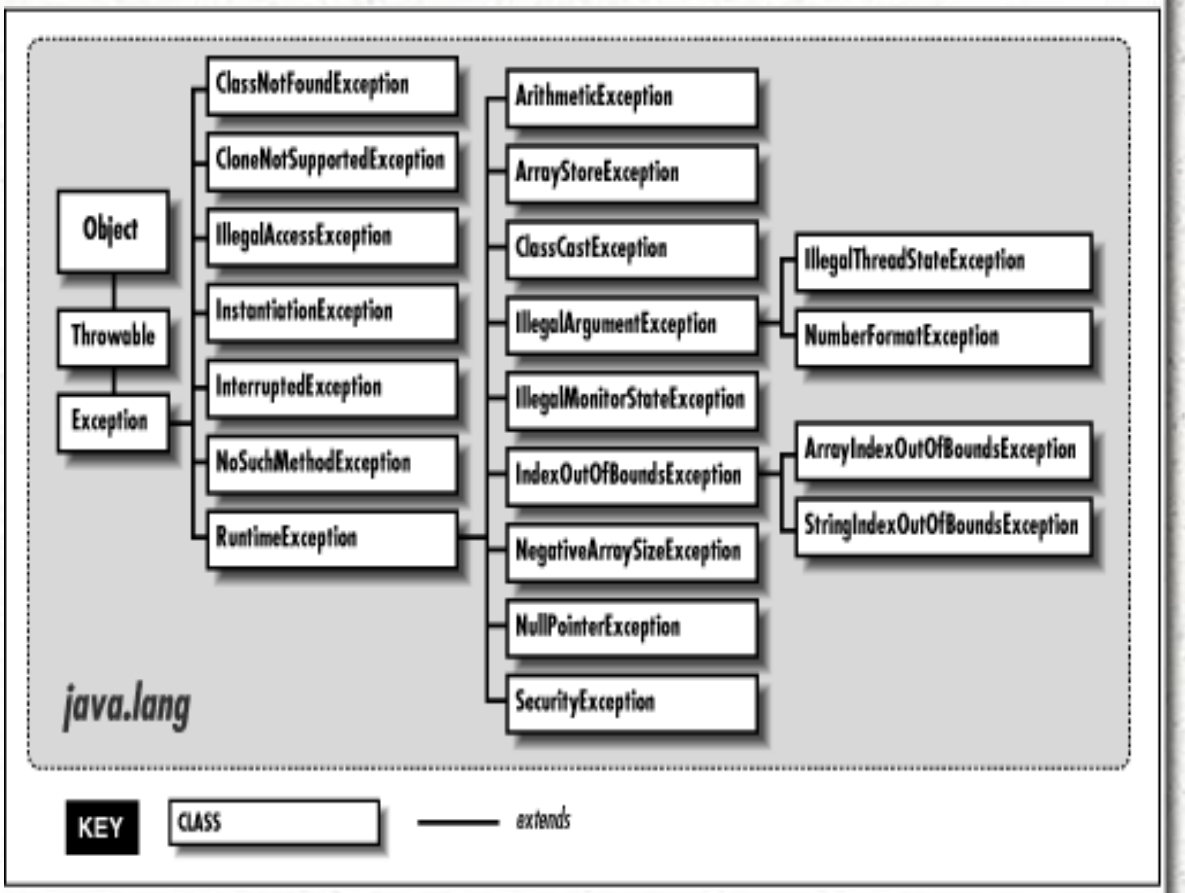
```
  case 1:    …  break;
  case 2:    …  break;
    :
  case n:    …  break;
  default:   …
}
```

Exceptions

- *Exceptions* are objects *thrown* when some unexpected condition occurs
  - All sub-classes of java.lang.Throwable
- Errors: Generally from JRE
  - java.lang.Error
- Exceptions: From JRE or user code
  - Checked: java.lang.Exception; must be on **throws** clause
  - Unchecked: java.lang.RuntimeException

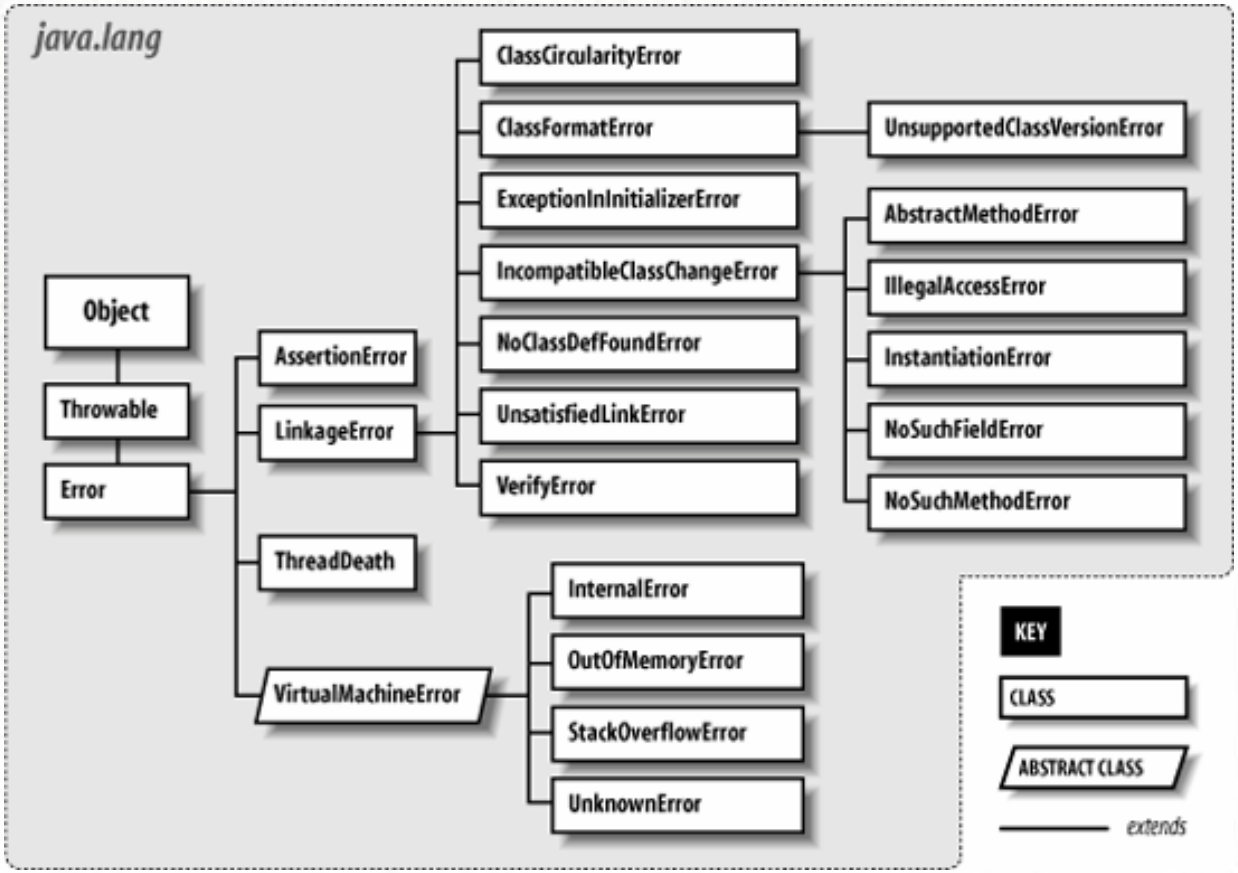
Exceptions

- Sample JRE Exceptions



Errors

- Sample JRE Errors



Throwing Exceptions

- Throw statement – throw an exception
- Throws declaration – list thrown exceptions

```
void doSomething()
    throws MyException, java.io.IOException
{
    // do something
    if(bad) throw new MyException();
    // do domething else
    if(broken) throw new
      java.io.IOException("cannot find file");
}
```

Exception Classes

- Must extend some other Exception
  - class MyException extends Exception
- Generally offer four constructors:
  - MyException()
  - MyExeption(Throwable cause)
  - MyException(String text)
  - MyException(String text, Throwable cause)

Try/Catch/Finally

- Use try/catch to capture exceptions
- Use try/finally to ensure resource cleanup
- Use try/catch/finally to do both

Try/Catch

- To capture exceptions
- Optionally consume exceptions
- Optionally recover from exceptions

```
try {
    :
    throw new Exception("it broke");
    :
} catch(Exception ex) {
    ex.printStackTrace();
    throw ex;
}
```

Try/Catch

- Can catch multiple types of exceptions

```
try {
    :
    throw new Exception("it broke");
    :
} catch(MyException me) {
   :
} catch(RuntimeException rte) {
   :
} catch(Exception ex) {
   :
}
```

Try/Catch

- If multiple catch have same code

```
try {
    :
    throw new Exception("it broke");
    :
} catch(MyException, RuntimeException me) {
   :
} catch(Exception ex) {
   :
}
```

Try/Finally

- Ensure resource cleanup/release

```
File file = File.open("somefile");
try {
    :
    throw new IOException("it broke");
    :
} finally {
  file.close();
}
```

Try/Catch/Finally

- Combines Try/Catch and Try/Finally

```
File file = File.open("somefile");
try {
    :
    throw new IOException("it broke");
    :
} catch(Exception ex) {
   :
} finally {
   file.close();
}
```

Try with resource

- Ensure resource cleanup/release

```
try (File file = File.open("somefile") {
    :
    throw new IOException("it broke");
    :
}
```

Similar behavior to prior slide

- Requires resource type to implement *Closeable*

Enums

- Sometimes you want constant values that are of arbitrary type, collectively named and can be tested with sameness (==)
- Enums often ints/strings, but can be other types
- Example: Enum of int

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

Enum Usage

- Test a value against enum cases

```
public class DayTest {

        public static void main(String[] args) {
                Day today = Day.SUNDAY;

                if(today == Day.SATURDAY || today == Day.SUNDAY) {
           System.out.printf("Yeah! %s is a weekend day!%n", today);
                } else {
           System.out.printf("Ugh! %s is another work day!%n", today);
                }
        }
}
```

```
Yeah! SUNDAY is a weekend day!
```

Enums

- More complex example: enum int w/values

```
public enum Planet {
    MERCURY (3.303e+23, 2.4397e6),
    VENUS   (4.869e+24, 6.0518e6),
    EARTH   (5.976e+24, 6.37814e6),
    MARS    (6.421e+23, 3.3972e6),
    JUPITER (1.9e+27,   7.1492e7),
    SATURN  (5.688e+26, 6.0268e7),
    URANUS  (8.686e+25, 2.5559e7),
    NEPTUNE (1.024e+26, 2.4746e7);

    private final double mass;   // in kilograms
    private final double radius; // in meters
    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }
    private double mass() { return mass; }
    private double radius() { return radius; }

    public static final double G = 6.67300E-11; // universal gravitational constant  (m3 kg-1 s-2)

    double surfaceGravity() {
        return G * mass / (radius * radius);
    }
    double surfaceWeight(double otherMass) {
        return otherMass * surfaceGravity();
    }

}
```

Annotations

- Meta-data applied to declarations
- Defined as “at interfaces” (@interface)
  - We will not define in this class, only use
- Standard and custom
  - Ex: @Deprecated void method() { … }
- Can have parameters: @name(“John”) ...

Testing

- An attempt to show a program is correct by using it and looking for aberrations/flaws
- A necessary evil
- Never 100% successful
  - All significant programs have bugs
- Always a time vs. completeness trade-off

How to Test

- Ad Hoc - as determined at test time
- Repeatable - pre-planned and codified
  - Tool - ex. jUnit
  - Scripted - sequence of steps humans do
- Philosophy
  - Test post-coding - traditional
  - *Test-Driven {Development}* (TDD) - popular in OO world

How to Test

- Manual - by people
- Semi-Automated - mostly by script/tool
- Fully Automated - all by script/tool

Types of Tests

- Unit - at unit (class/module) level
- Component - at component (related classes, say package)
- System - all components together
  - Systems of systems possible
- Function - externally visible function
- Regression - to detect flaws due to changes

Types of Tests (cont)

- Performance - to verify meets performance requirements
- Usability - to verify meets ease-of-use requirements
- etc. - many more types possible

Test Approach

- Black-Box
  - Cannot see into code
  - Specification driven
  - Function, System, etc.
- White-Box
  - Code internals exposed
  - Unit, Component

jUnit

- Test Programs using annotated code
  - –@Xxxx annotations on code
  - –assertTrue(price == 1.0)
- Run with Junit Runner
  - –java org.junit.runner.JUnitCore TestClass1 [...other test classes...]

jUnit Annotations

- @Before – runs before each @Test method
- @BeforeClass – runs before first @Test method
- @After – runs after each @Test method
- @AfterClass – runs after last @Test method
- @Test – marks a method as a test
  - @Test(timeout=500) – can have constraints
  - @Test(expected=IllegalArgumentException.class)
- @Ignore – skips this method

jUnit Example

- Simple class under test

```
public class Subscription {
   private int price;  // in euro-cent
   private int length; // subscription months

   public Subscription(int price, int months) {
     this.price = p; this.length = months;
   }

   // Calculate the monthly subscription price
   // in euro
   public double pricePerMonth() {
     return (double)price / (double)length;
   }

   // Cancel subscription.
   public void cancel() { length = 0; }
}
```

jUnit Testcase

- Test class
- Often more test code than tested code

```
import org.junit.*;
import static org.junit.Assert.*;

public class SubscriptionTest {
   @Test
   public void test_returnEuro() {
      System.out.println(
"Test if pricePerMonth returns Euro...");
      Subscription s = new Subscription(200,2);
      assertTrue(s.pricePerMonth() == 1.0);
   }
   @Test
   public void test_roundUp() {
      System.out.println(
 "Test if pricePerMonth rounds up correctly...");
      Subscription s = new Subscription(200,3);
      assertTrue(s.pricePerMonth() == 0.67);
   }
}
```

Requirements

- Business
  - Cost
  - Schedule
- Functional
- Performance
- Code-base Size
- Usability

Requirements (cont)

- Maintainability
- Logistical

–Developer/Tester skills

–Language choice

–Platform choice

–etc.

Congratulations!