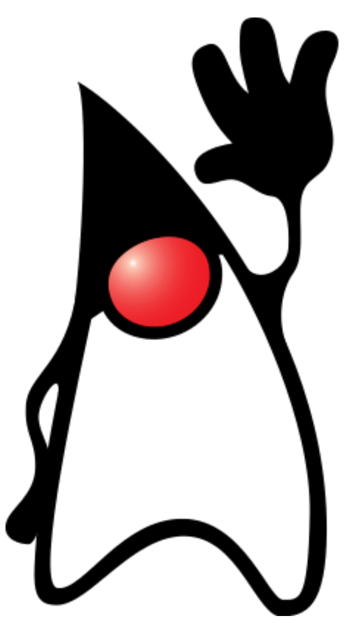
# COMP 150-SEN
# Software Engineering Foundations

## Introduction to Java

Spring 2019

# Introduction

- Java is a general-purpose, object-oriented programming language
  - Java 1.0 released in 1996
  - Initially (well before release) called Oak
- Key goal: Write once, run anywhere
  - And it's even sort of true
- Key early idea: Run Java in your browser
  - But when's the last time you ran an "applet"?
  - Probably, never



# Warnings

- Java looks C/C++-ish, **but it's not**
  - The semantics of Java are very different
  - Example: No *p in Java
  - Example: Java memory is not a big array of bytes
- Object-oriented (OO) programming in Java is clean, OO programming in C++ is a minefield
  - If you go back to C++, be very careful
  - Avoid using C++ features just because they are there
  - Use as minimal a subset of C++ as possible
- JavaScript has nothing to do with Java
  - JavaScript has objects, but they are wacky
  - BTW, the standards body calls JavaScript "ECMAScript"

# Hello, world.

A.java

```
public class A {
    public static void main(String[] args) {
     System.out.println("Hello, world.");
    }
}
```

- Compile with `javac A.java`
  - This will produce a file `A.class`, called a *class file*
    - Contains *java bytecode*, see it with `javap -c A`
  - Every class should be in a `.java` file with the same name
    - (Not strictly enforced but is standard practice, and IDEs assume it)
- Run with `java A`
  - This starts the program at `A`'s `main` method

# Java Virtual Machine

A.java

```
public class A {
    public static void main(String[] args) {
     System.out.println("Hello, world.");
    }
}
```

- Class files contain *java bytecode*, **not** machine code
  - See it with `javap -c A`
- Programs are run inside the *Java virtual machine*
  - In simple case, this is an *interpreter*
    - Reads bytecode instructions one by one and executes them
  - That would be slow, so Java has a *just-in-time compiler*
    - Compiles “hot” methods (ones called often) to machine code
    - But this will not matter to you for this class

# Classes and Methods

A.java

```
public class A {
    public static void main(String[] args) {
     System.out.println("Hello, world.");
    }
}
```

- Java (in fact, OO) functions are called *methods*
  - We'll see why they have a different name shortly
  - This program has one method, `main`
- Methods live inside of classes
  - This program has one class, `A`
  - And a class is a collection of methods
- Java Program = a set of classes
  - Invoking `java C` will start `C`'s `main` method

# Naming

- Good names are a precious commodity
  - Later, we'll have a lecture about what makes names good
  - Important Java rules
    - Class names are capitalized
    - Method and variable names are not capitalized
- So we often want to reuse the same name in different contexts
  - How many times do you use `i`, `j`, `x`, `y` to mean different things in the same program?
  - More complex names are also often reused

# Scoping

- Languages use *scopes* to
  - Bind names to meanings
    - E.g., `int i`; `char *p`; etc
  - Allow the same name to be used to mean different things in different scopes
  - (Aside: C, C++, and Java all use *static scoping*; see 105)
- One scope indicator you know: `{}`'s

a.c

```
int add1(int x) {
  int y = x;  // x and y can only be used in add1
  return y + 1;
}
int first(char *y) {
  char x = y[0]; // (let's assume y not NULL)
  return x;    // x and y reused in first, differently
}
```

# C Has a Flat Namespace

a.c

```
void foo() {
  int x;  // x can only be used in foo
  …
}
void bar() {
  float x;  // x can be reused, differently, in bar
  …
}
```

- But foo and bar are both in the extern scope
- They will be even if we put them in different files!
  - Assuming we link those files together
- What if we want to reuse code from two projects that use different functions with the same name?!
  - Have to go through and rename…ugh…

# Classes as Namespaces

Arith.java

```
class Arith {
  public static int add1(int x) { return x+1; }
  public static int add2(int x) { return add1(add1(x)); }
}
```

```
int a = Arith.add1(2);      // returns 3
int b = Arith.add2(2);      // returns 4
```

- Here `static` indicates a *class method*
  - Inside the class, referred to by method name alone
  - Outside the class, referred to as `Class.method`
    - Sometimes called *the dot notation*
- Warning: `static` means something different in C
  - In fact, it means several different things depending…

# Classes as Namespaces (cont'd)

Arith.java

```
class Arith {
  public static int add1(int x) { return x+1; }
  public static int add2(int x) { return add1(add1(x)); }
}
```

Mod3Arith.java

```
class Mod3Arith {
  public static int add1(int x) { return (x+1)%3; }
  public static int add2(int x) {return(add1(add1(x)))%3;}
}
```

```
int a = Arith.add1(2);        // returns 3
int b = Arith.add2(2);        // returns 4
int c = Mod3Arith.add1(2);    // returns 0
int d = Mod3Arith.add1(0);    // returns 1
```

# Shadowing

- Occurs when something of same name declared in an inner scoping, hiding name from outer scope

```
public static int foo(int x) {
  char x = ‘c’; // shadows “int x”
  double x = 42.1; // shadows “char x”
 { int x = 3; // shadows “double x” }
}
```

- Java disallows all three cases
  - Theory: Shadowing is an *anti-pattern* or a *code smell*
  - Something that might not be wrong, but often is or often leads to mistakes later
- Java does allow fields to be shadowed by local vars
  - We’ll learn what fields are shortly

# Java Method Definition Order

Arith.java

```
class Arith {
  public static int add2(int x) { return add1(add1(x)); }
  public static int add1(int x) { return x+1; }
}
```

- `add1` called before definition
  - Okay in Java (but not in C!)
    - (Why? Because C designed for a *single pass compiler*)
- Methods may appear in any order
  - Within a method, declaration order rules same as C

```
public static int foo(int x) {
  int a = x;   // okay
  int b = c + 1;   // error
  int c = 0;
}
```

# Primitive Types, Variable Decls

- Java *primitive types* are similar to C

```
public static int foo(int x, double y) {
  byte d = 81;          // 8 bits
  short e = 42;         // 16 bits
  int a = x + 2;    // 32 bits
  long f = 423874289; // 64 bits
  float b = 3.14f;  // typically not used
  double c = y - 42.1;
  char d = ‘a’;
  boolean b = true;
}
```

- Methods return `void` to indicate no interesting value
  - No-argument methods have empty arg lists (not `void`)

```
public static void useless() { return; }
```

# Java Control Structures

- Java conditionals and loops look mostly like C, except guard must be a `bool`

```
if (x == 5) {     // ()'s required
  y = 2;          // use {}'s to avoid pain later
} else {
  y = 3;
}
```

- C-like `while` loops and `for` loops

```
while (x < 5) { x++; }
for (int i=0; i<5; i++) { x--; }
```

# Java Switch Statement

- Just like in C

```
switch (x) {
  case 0:
    y = 2;
    break;      // exit switch statement
  case 1:
    y = 3;      // no break, falls through
  case 4:
    z = 2;
  default:
    y = 42;     // if no prior case matches
}
```

# Abstract Data Types (ADTs)

- An *abstract data type* is some data along with a collection of operations on it
  - One of the fundamental building blocks of good code
  - We'll talk about where "abstract" comes from in a bit

# A Point ADT in C

```
#include <math.h>
#include <stdlib.h>
typedef struct point { int x; int y; } *point;
point mkPoint(int a, int b) {
  point p = malloc(sizeof(*point)); p->x = a; p->y = b;
  return p;
}
point shift(point p, int dx, int dy) {
  return mkPoint(p->x + dx, p->y + dx);
}
double dist(point p1, point p2) {
  return sqrt(pow(abs(p1->x - p2->x), 2) +
              pow(abs(p1->y - p2->y), 2));
}
point p1 = mkPoint(1, 5);
point p2 = mkPoint(2, 10);
point p3 = shift(p1, -1, 5);
double d = dist(p2, p3);
```

# ADTs are Awesome

- Key idea: *Information hiding* (or *abstraction*)
  - `point`s come with an *interface* for working with them
    - Interface = set of functions provided for `point`s
  - Code that uses interface is oblivious to *implementation*
    - Doesn't need to know that points are represented as (x,y) coordinates
- Two key benefits
  - Primary: Client doesn't need to understand implementation
    - E.g., client doesn't need to know how to compute distance
    - Many programmers can work on same program without conflicts!
      - (As long as they agree on the interface)
  - Secondary: Implementation can *change* without modifying clients
    - E.g., could switch to polar coordinates

# Barbara Liskov

- ACM Turing Award 2008
  - For contributions to practical and theoretical foundations of programming language and system design, **especially related to data abstraction**, fault tolerance, and distributed computing.

- CLU programming language
  - B. Liskov and S. Zilles. Programming with Abstract Data Types. ACM Sigplan Conference on Very High Level Languages. April 1974

# Java Fields and Constructors

- Classes can also be used to store data in *fields*

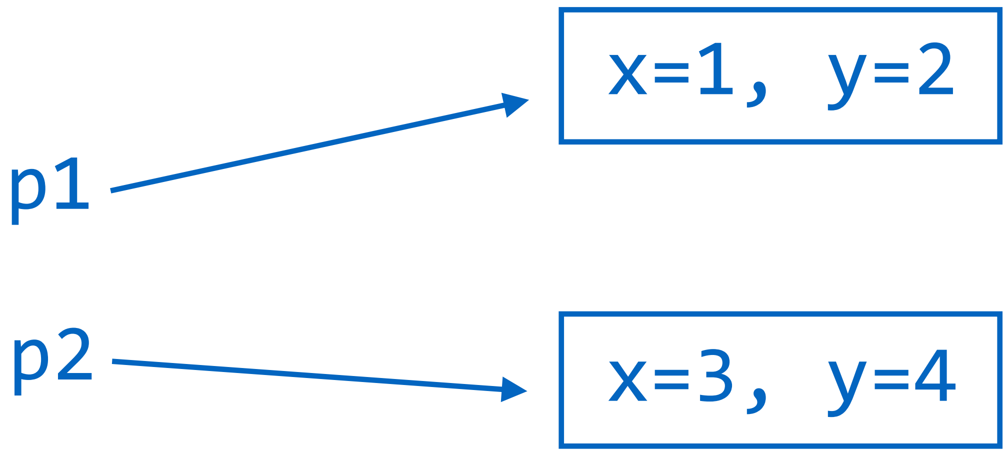
```
class Point {
  private int x; // x and y are fields
  private int y;

  Point(int a, int b) { // constructor
    this.x = a;
    this.y = b;
} }
Point p1 = new Point(1, 2);
Point p2 = new Point(3, 4);
```

  - Writing `new Class(args)`
    - Allocates a new *object* of the right size
    - Calls the constructor to initialize it (we'll talk about >1 constructors later)
    - Returns the newly allocated and initialized memory

# Java Objects

```java
Point p1 = new Point(1, 2);
Point p2 = new Point(3, 4);
```




- The object `new C` returns is an *instance* of class C
  - `p1` and `p2` are both instances of `C`
- Remember, classes can also have methods
  - OO programming = putting fields and methods together
    - (Aside: the idea of OO programming (OOP) doesn't strictly require classes, but classes are the most common approach to OOP)

# A Point ADT in Java (Class Meths)

```
class Point {
  private int x; private int y;
  Point(int a, int b) { this.x = a; this.y = b; }
  public static Point shift(Point p, int dx, int dy) {
    return new Point(p.x + dx, p.y + dy);
  }
  public static double dist(Point p1, Point p2) {
    return Math.sqrt(Math.pow(Math.abs(p1.x-p2.x), 2) +
                     Math.pow(Math.abs(p1.y-p2.y), 2));
  }
}
Point p1 = new Point(1, 5);
Point p2 = new Point(2, 10);
Point p3 = Point.shift(p1, -1, 5);
double d = Point.dist(p2, p3);
```

# This is Everywhere

- The code on the previous slide isn't much better than C…yet…
- But notice something that happens with ADTs
  - Every method of the ADT takes at least one of ADT instance as an argument
  - Which makes sense because the methods are there to manipulate the ADTs…
- OO programming has special support for this pattern

# Instance Methods Example

```
// a class method
class Point { // fields and constructor as before
 public static int getX(Point p) { return p.x; }
}
Point p1 = new Point(1, 5);
int x = Point.getX(p1);
```

```
// an instance method
class Point { // fields and constructor as before
 public int getX() { return this.x; } // not static
}
Point p1 = new Point(1, 5);
int x = p1.getX();
```

# Instance Methods

- An *instance method* is a method not defined `static`
- Every instance method has a special argument this
  - Must be referred to as `this` inside method body
  - (Not possible to redefine `this` inside method)
  - When passed to the method:
    - Placed to the left of the method name followed by a dot
      - `classMethod(obj, arg1, arg2, …)` becomes
      - `obj.instanceMethod(arg1, arg2, …)`
    - Omitted from list of formal parameters
      - `public static type meth(obj, arg1, arg2, arg3, …)` becomes
      - `public type meth(arg1, arg2, arg3, …)`

# A Point ADT in Java (Inst Meths)

```
class Point {
  private int x; private int y;
  Point(int a, int b) { this.x = a; this.y = b; }
  public Point shift(int dx, int dy) { // not static
    return new Point(this.x + dx, this.y + dy);
  }
  public double dist(Point p2) { // not static
    return Math.sqrt(Math.pow(Math.abs(this.x-p2.x), 2) +
                     Math.pow(Math.abs(this.y-p2.y), 2));
  }
}
Point p1 = new Point(1, 5);
Point p2 = new Point(2, 10);
Point p3 = p1.Point.shift(-1, 5);
double d = p2.Point.dist(p3);
```

# This Can be Omitted

- You can omit `this` when referring to fields
  - Just be careful because local variable names can shadow field names
  - Sometimes need to use `this` to disambiguate

```
class Point {
  private int x; private int y;
  void setX(int x) {
   this.x = x;   // is this bad style? unclear
} }
```

# Public, Private

- Public methods and fields are visible from outside the class
- Private methods and fields are only visible inside the class

```
class Demo {
  private int x; public int y;
  int sum() { return x+y; }  // okay
}
Demo d = …;
int a = d.x; // error
int b = d.y; // okay
```

- Notice we made fields private in `Point`
  - Enforces information hiding!

# Public, Private Tips

- Generally, make all fields private
  - You don't have to, but it can save you pain later on
  - Ensures no other code can mess with an object's fields
- If you want to make fields public, consider getters and setters

```
class Demo {
  private int x;
  public int getX() { return this.x; }
  public void setX(int x) { this.x = x; }
}
```

  - Allows you to intercept access to fields to enforce invariants (for setters) or transform/hide certain data (getters)
    - Ex: setter - make sure `null` never stored in data structure
    - Ex: getter - for web server, check current user has access to data

# Public, Private Tips (cont'd)

- Generally, make most methods public
  - "Helper" methods are a good candidate for `private`
    - Methods that other methods could reasonably use, but that methods outside the class have no business invoking
    - Ex: A hash table implementation might have a `resize` method that allocates more memory for the table, which might be called in a few places

- You can leave off `public` and `private`
  - This means "mostly public," but the exact semantics are probably not what you want
  - Best to get in the habit of marking methods explicitly

# What's the Point of This?

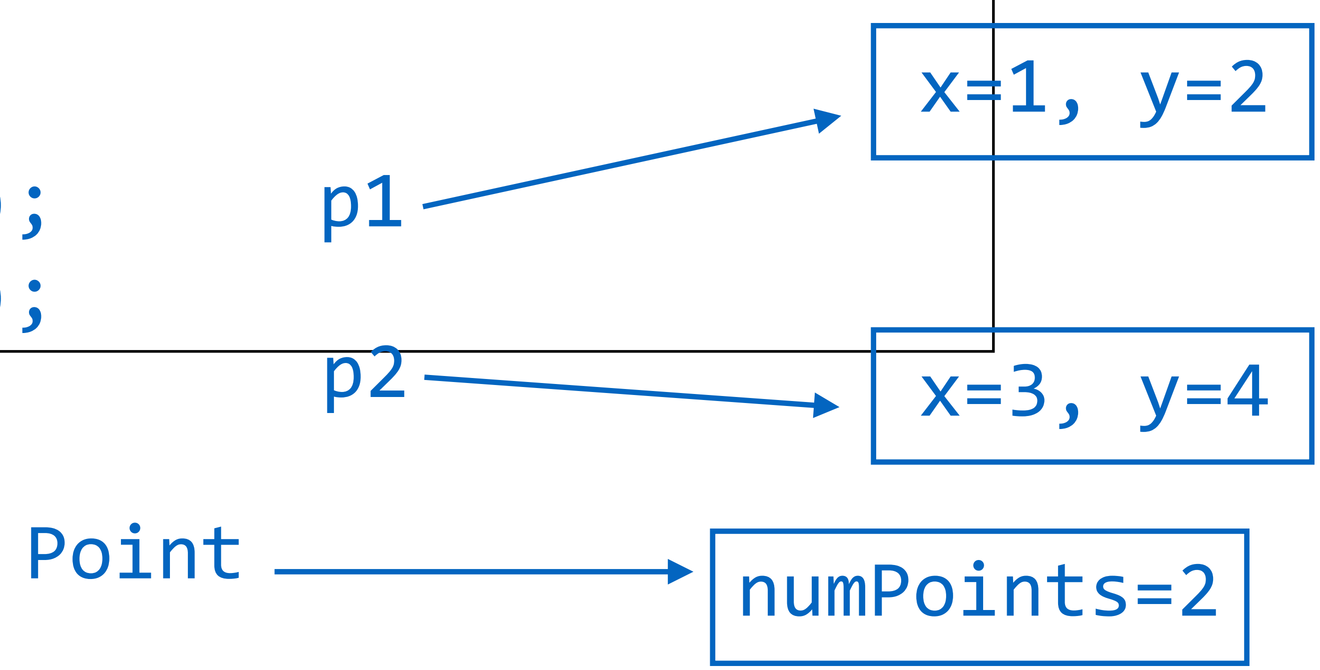
- Recall some key SE properties: correctness, maintainability
  - Often comes down to code that is *easy to understand*
  - Using objects *sometimes* results in cleaner code
    - (Note: If anyone tells you that one particular programming paradigm is the ultimate solution to writing good code, you can laugh in their face)
- OOP has a few potential advantages
  - Using *objects* groups code and data together is a common pattern (improves code often, not always)
    - public/private a big win
  - Support for *dynamic dispatch* eliminates some conditionals, and can improve code (sometimes)
  - Support for inheritance can reduce code duplication, thus improving code (very, very rarely)

# Class Fields

- C*lass fields* shared across all instances of the class
  - A global variable, but slightly less global

```
class Point {
  private int x, y;
  private static int numPoints; // class field

  Point(int a, int b) {
    this.x = a; this.y = b;
   numPoints++;
} }
Point p1 = new Point(1,2);
Point p2 = new Point(3,4);
```



# Hello, world. Redux

A.java

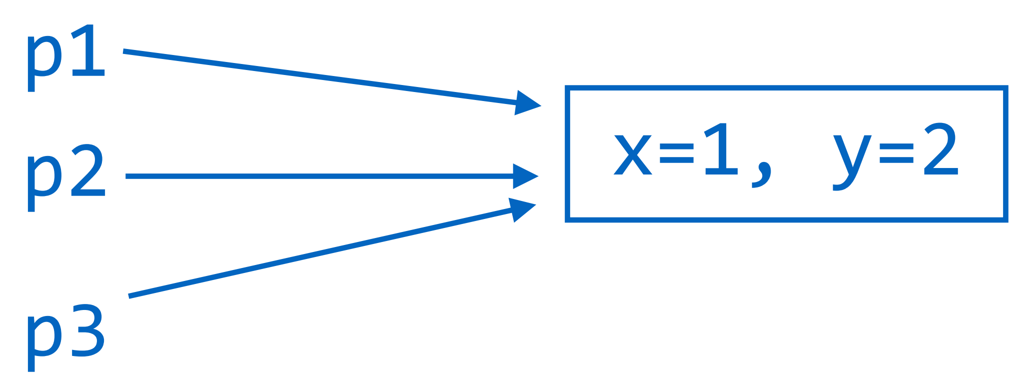
```
public class A {
    public static void main(String[] args) {
     System.out.println("Hello, world.");
    }
}
```

- Look at the call to `println` line carefully
  - It reads the class field `System.out`
  - That returns an object (which represents stdout)
  - It invokes that object’s instance method `println`
- Not so mysterious any more!

# Objects are Pointers

- An object in Java is actually a *pointer* or *reference* to the object in the heap
  - You don't need to mess around with `*`'s and `->`'s because objects are *always* pointers
  - Copying or passing an object means copying the pointer value

```
p1 = new Point(1,2);
p2 = p1;
foo(Point p3) { … }
foo(p1);
```




- If you want to make a *deep copy*, need to do so explicitly
  - Warning: Never use the `clone` method, it makes a copy one level deep, which is almost never what you want

# Physical vs. Structural Equality

- Java's `==` method compares objects by *physical equality*, i.e., it compares pointers

```
p1 = new Point(1,2);
p2 = p1;
p1 == p2; // true
p3 = new Point(1,2);
p1 == p3; // false
```

- *Structural equality* is checked by .equals
  - Have to implement this yourself!

```
class Point {
  boolean equals(Point p) {
    return x==p.x && y==p.y;
} }
p1.equals(p3)  // true
```

# Strings are Objects

- Java primitives (`int`, `double`, etc) are not objects
  - Can't invoke methods on them
- Java `String`s are objects
  - Can be created without `new` by writing down a string literal
  - Thus, they have methods!

```
s1 = "Hello,"
s1.length();           // 6
s2 = " world.\n";
s3 = s1.concat(s2);    // s3 = "Hello, world.\n"
```

  - See Java API documentation for method list
    - https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/String.html

# Tips on Equality

- Much of the time, `==` is a performance optimization
  - And remember, premature optimization is the root of all evil
- Use `.equals` unless it's too expensive and you're sure `==` is safe
  - Most common safe case: *immutable objects* whose allocation is tightly controlled
  - Almost an example: Java `Strings` are immutable
    - But it's not guaranteed that the same string will always be represented by a single object

B.java

```
class B {
  public static String oo() {
    return "oo"
} }
```

A.java

```
"foo" == "foo"   // true
"foo" == "f" + "oo" // true
"foo" == "f" + B.oo() // false
```

# null

- Java's null pointer is called `null`
- `null` can be used wherever an object is expected
- Error to invoke method or access field of `null`

- Enough said?

- Maybe not: Tony Hoare called the null pointer his "Billion dollar mistake"
  - Why are null pointer errors so pernicious?
    - Good property: they abort immediately with an error
      - Helps with debugging
    - Bad property: compilers don't do much to prevent them

# Garbage Collection (GC)

- You've seen how to allocate objects with `new`, but you do you free them?
  - Answer: you don't!

- Java has (*automated) garbage collection*
  - Every once in a while, the JVM finds "dead" objects and frees them for you
    - Ideally, dead = objects that will never be used again
    - Actual algorithm, dead = object not reachable from the stack and class fields
      - If the program can't follow a chain of zero or more pointers to the object starting from local variables or class fields, the object must be dead
- See COMP ?? to learn more about GC

# Inner Classes

- Java classes can be nested
  - A class that is inside another is an *inner class*

```
class A {
  static class B { // B instances not linked to A's
    void f() { System.out.println("B.f!\n"); }
  }
  void test() {
    b = new B();
    b.f()
} }
```

  - Note: `B` is compiled to `A$B.class`
    - The JVM doesn't know about inner classes; they are *syntactic sugar*
      - "Syntactic sugar causes cancer of the semicolon." — *Alan Perlis*
  - There's a bunch of trickiness with inner classes, but for now the above is all you need to know

# Example: LinkedList

- Exercise: Using what we know to implement linked lists
  - To signal errors correctly, do need to get ahead a little bit and use an exception
  - We'll see these in detail later
- See 02-LinkedList.java

# Java Arrays

- Built-in to Java, syntax similar to C, but…
  - Arrays know their length
  - Not possible to read/write out of bounds

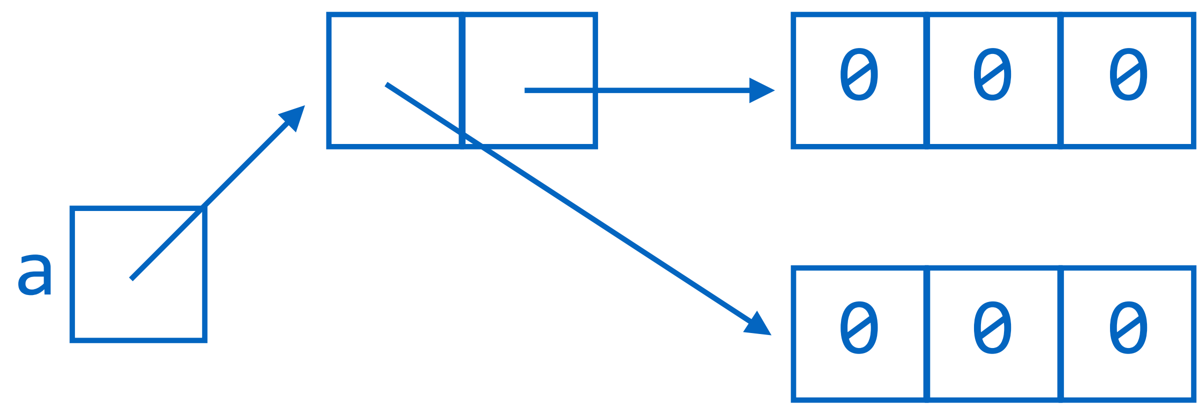
```
int[] a = new int[5];  // array of five ints
                       // initially, all 0 (default val)
a[0] = 42;
int x = a[0];
a[5] = 43;             // runtime error
int y = a[-1];         // ArrayIndexOutOfBoundsException
```

- Use `a.length` to get length of array `a`
  - Not quite a method (notice no ()'s)!
- (`Array`s are objects)

# Multidimensional Arrays

- Multidimensional arrays work fine in Java
  - But they are not necessarily stored contiguously

```
int[] a = new int[2][3];
```



# Example: ArrayList

- Exercise: Using what we know to implement lists using arrays
  - Since arrays are fixed size, need to resize if we try to insert too many elements
- See 02-ArrayList.java

# Functional vs. Non-Functional Requirements

- We’ve now implemented the same ADT two ways
  - `ArrayList` and `LinkedList` satisfy same *functional requirements*
    - What the code is supposed to *do*
    - Often, input-output behavior
  - But they differ in *non-functional* attributes
    - Things like: security, reliability, performance, maintainability, scalability, and usability
      - (Don’t spend too long worry about why these are non-functional requirements; think of this as a terminology choice)
    - ArrayList and LinkedList have different performance profiles

| $n$ = length | `get(int)` | `pop()` |
| --- | --- | --- |
| `ArrayList` | $O(1)$ | $O(n)$ |
| `LinkedList` | $O(n)$ | $O(1)$ |

# Java Interfaces

```
interface List {
  void insert(int x);
  int size();
  int get(int pos);
  int pop();
}

class ArrayList implements List { … }
class LinkedList implements List { … }
List l1 = new ArrayList();
List l2 = new LinkedList();
l1.insert(42); …
```

# Java Interfaces (cont'd)

- Describe publicly visible methods in classes
- A class that `implements` an interface must have at least the methods in the interface
  - Okay to have more methods, both public and private
- If class `C` implements `I`, then a `C` can be used where an `I` is expected
  - This is called *subtyping* or *subtype polymorphism*
- Interfaces must be specified explicitly
  - If `C` has methods of interface `I` but doesn't implement `I`, then `C` *cannot* be used as an `I`
- Classes can implement more than one interface
  - Comma-separated list after `implements`

# Interfaces and Info. Hiding

- Client that uses interface can only refer to interface methods, not other class methods

```
interface I {
  void m1();
}
class C implements I {
  public void m1();
  public void m2();
}

I i = new C();
i.m1();  // okay
i.m2();  // error
```

# Dynamic Dispatch

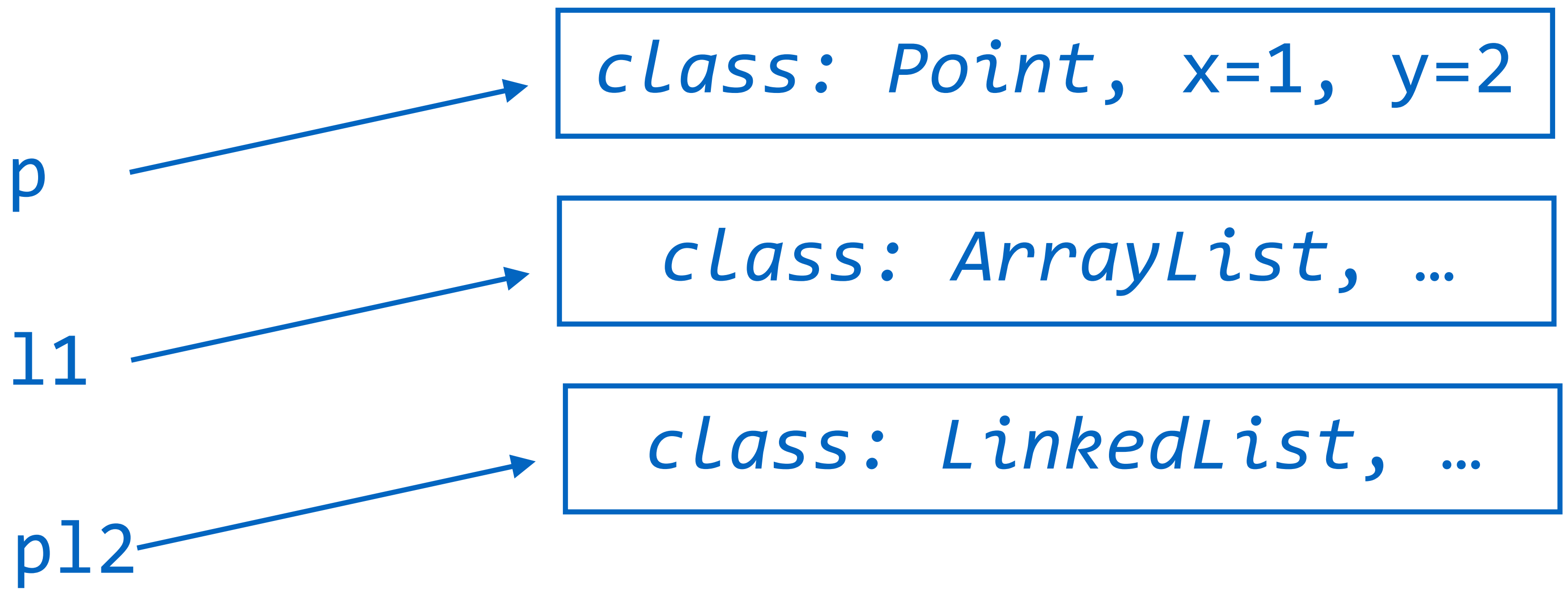
```
List l;
if (some complex condition)
   l = new ArrayList();
else
   l = new LinkedList();
l.insert(42); …
```

- How does JVM know which method to call?
  - Answer: Uses run-time information
- Key to OOP: dynamic dispatch
  - At a call `o.m(arg1, …, argn)`
    - Look up the run-time type of `o`, i.e., what class is `o` an instance of?
    - Invoke that class's `m` method
  - `o` is called the *receiver*, and this is why it's treated specially

# Run-Time Types

- Every Java object knows its type at run time

```java
Point p = new Point(1, 2);
List l1 = new ArrayList();
List l2 = new LinkedList();
```




  - Note: *class* is **not** a real field that you can access directly, it’s just made up for illustration purposes

# Instanceof and Type Casts

- Possible to test types directly, but discouraged

```
List l = …;
if (l instanceof ArrayList) {
  al = (ArrayList) l;
  // use al here
} else if (l instanceof LinkedList) {
  ll = (LinkedList) l;
  // use ll here
}
```

- Unlike C, type cast `(C) e` is checked
  - Fails at run time if `e` does not evaluate to a `C`
  - Some “useless” casts generate compiler error
    - If they would always fail at run time

# Instanceof is Often Bad Style

- Must know all possible implementors of `List`
  - If we add another kind of `List`, need to modify this code!
- Also more verbose
  - Dynamic dispatch has `if` built-in
- If you find yourself writing this kind of code, perhaps the interface is not the right choice

- But, sometimes it is the right design
  - Most common case: implementing ML-style pattern matching in OOP
    - Which is icky no matter how you do it
    - Alternative: *visitor pattern*, which we will see later

# Packages

- *Package* = set of classes
- Created with a `package` declaration

```
package edu.tufts.edu;
class C { … }
```

  - Note: files in packages need to be in certain directory struct.
- Classes in packages accessed by package name

```
new edu.tufts.cs.C();
```

- Packages may be *imported* into current namespace

```
import edu.tufts.cs;
new C();
```

  - `import edu.tufts.*;` for all packages beginning with `edu.tufts`
  - IDEs often import packages automatically

# Java Standard Library

- *Library* = collection of classes and methods to be called by your program
  - Ex: String manipulation, I/O, networking, cryptography, etc.
  - Might come from third-party, or one part of a big program might be considered a library
- Libraries are more important than the language!
  - Java was one of the first languages to figure this out
  - Much of modern coding is figuring out how to use libraries
- For Java library, see JDK 11 API on class web page
  - Look under `java.base`, especially
    - `java.lang` - basic language stuff, package always open
    - `java.util` - collections (lists, etc.)
    - `java.io` - file access

# Inheritance (Extends)

- Warning: ***Inheritance is almost always a bad idea***
  - But it's so deeply embedded in common OO language design you need to know about it
- Goal of inheritance: *code reuse*
  - A worthwhile goal!
  - Code reuse is what makes big software possible
- Examples of code reuse?
  - Functions/methods ("subroutines")
  - Libraries
  - Frameworks
  - Stack Overflow?

# Superclasses and Subclasses

- In OOP, a class *inherits* methods from its *superclass*

```
class A {
  void m() {
    System.out.println(“A.m”);
} }
class B extends A { }

(new A()).m()   // prints “A.m”
(new B()).m()   // prints “A.m”
```

  - If `B` extends `A`, then all methods of `A` are also added to `B`
  - `A` is the *superclass*, `B` is the *subclass*
- Inheritance is transitive, e.g., if `C extends B` and `B extends A`, then `C` has all methods of `B` and `A`

# Subclasses Can Have More Meths

```
class A {
  void m() {
    System.out.println("A.m");
} }
class B extends A {
  void p() {
    System.out.println("B.m");
} }

(new B()).p()   // prints "B.m"
(new A()).p()   // error
```

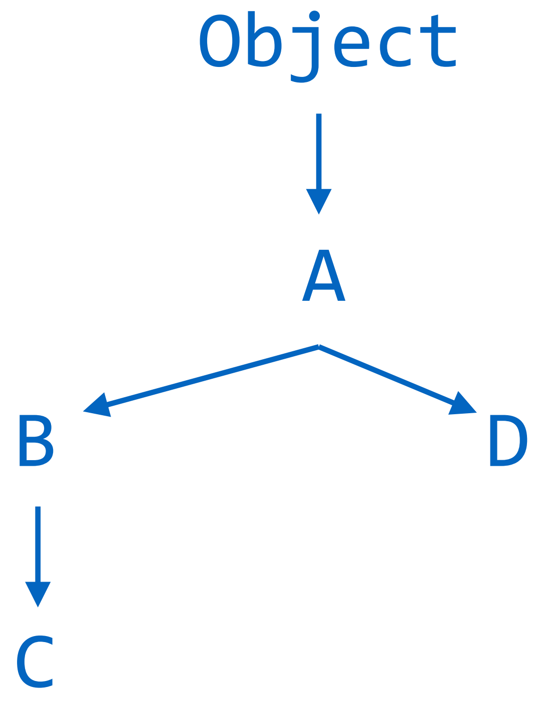
- Subclasses can have more methods than superclasses

# Class Hierarchy

- Every class has exactly one superclass
  - Java does not have *multiple inheritance*
  - (C++ does support multiple inheritance)
- If a class doesn't have an `extends` clause, then it `extends` the special class `Object`
- Thus, we can draw a tree where a superclass is the parent of its immediate subclass
  - This is the *class hierarchy*, and `Object` is the root
- Thus, the set of classes forms a tree, with `Object` as the root

# Example Class Hierarchy

```
class A { … }
class B extends A { … }
class C extends B { … }
class D extends A { … }
```



# Useful Methods of Object

- `boolean equals(Object obj)`
  - Indicates whether some other object is "equal to" this one
  - Default: physical equality
- `int hashCode()`
  - Returns a hash code value for the object
  - Default: roughly object's address in memory
- `String toString()`
  - Returns a string representation of the object
  - Default: `getClass().getName() + '@' + Integer.toHexString(hashCode())`
- (From https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html)

# Overriding Superclass Methods

- Sometimes subclasses want a method that behaves differently than a superclass method

```
class A {
  void m() {
    System.out.println(“A.m”);
} }
class B extends A {
  void m() {
    System.out.println(“B.m”);
} }

(new A()).m()   // prints “A.m”
(new B()).m()   // prints “B.m”
```

# Another Overriding Example

```
class Rectangle {
  int area() {
    return length * width;
} }

class Square extends Rectangle {
  int area() {
    return length * length
} }
```

# Good Uses of Overriding

- Overriding methods of `Object`!

- `boolean equals(Object obj)`
  - Change to deep equality
- `int hashCode()`
  - Invariant: equal objects should have equal has codes!
  - Quick sanity check: Any class that overrides `equals` should override `hashCode`
- `String toString()`
  - Print something useful

# Example: Point

```
class Point {
  private int x; private int y;
  boolean equals(Point p) {
    return x == p.x && y == p.y;
  }
  int hashCode() {
    return 31 * x + y;
} }
```

- Need
  - `(new Point(1,2)).equals(new Point(1,2))`
  - `(new Point(1,2)).hashCode() == (new Point(1,2).hashCode())`

# Super

- Sometimes subclass method wants to override a method from a superclass, but use the superclasses's method

```
class A {
  void m() {
    System.out.println("A.m");
} }
class B extends A {
  void m() {
    super.m();
    System.out.println("B.m");
} }

(new A()).m()    // prints "A.m"
(new B()).m()    // prints "A.m\nB.m"
```

# Why is Inheritance a Bad Idea?

- Subclass and superclass often tightly *coupled*
  - Changing one forces changes to other, which forces changes other sub/superclasses, etc
- Superclass methods pollute namespace of subclass
  - This is why Java does not have multiple dispatch
    - If `C extends A` and `B`, and both `A.m` and `B.m` are defined, what is `C.m`?
  - But then limits reuse to only one superclass
- Class hierarchies are brittle
  - Often subclass/superclass relationships make sense for one *concern* but not for another
    - A *concern* is some piece of concept of functionality of the code, e.g., logging, printing messages, storing data, etc
- Soln: Prefer *delegation* to inheritance
  - More on this later

# Inheritance vs. Delegation

```
class Rect {
  private int width, height;
  public int area { return width * height; }
}
```

- Compare:

```
class Window extends Rect { … }
```

```
class Window {
  Rect r;
  public int area { return r.area(); }
}
```

  - The lower example uses *delegation*
    - It delegates the area method to `Rect`
    - Black box reuse! Don't need to know insides of `Rect`
    - Warning: most examples of inheritance look sensible but are not

# Wrapper (Boxed) Classes

- Primitives are not objects
  - `int`, `boolean`, `double`, etc
- Good for performance, bad for flexibility
  - E.g., can't make a `List<int>`
- Solution: "boxes" for primitives
  - `Integer`, `Boolean`, `Double`, etc
  - `Integer.valueOf(42);` *// returns 42, boxed*
  - `Integer.valueOf(42).intValue();` *// returns 42*
- Java includes *autoboxing*
  - Will try to insert conversions from primitives to boxed objects where necessary

# Preconditions

- Functions often have requirements on their inputs
  - These are called *preconditions*

```
// Return maximum element in a[i..j]
int findMax(int[] a, int i, int j) { … }
```

- Possible preconditions?
  - `a` is non-empty
  - `i` and `j` must be non-negative
  - `i` and `j` must be less than a.length
  - `i < j` (maybe)
- What should a method do if precondition isn't met?

# Returning Invalid Values

- One approach: Return a value that is outside the range of possible valid returns

```
// Returns a value key maps to, or null if no
// such key in map
Object get(Object key)
```

- Several disadvantages
  - Caller needs to remember to check for erroneous result
  - Caller needs to handle result immediately
    - What if some method further up the call chain is the right place to handle the error?
  - Caller can't distinguish multiple different error conditions
  - Requires valid returns to be sub-range of return type

# Error Status Codes

```
// From an ancient version of Linux
static int lock_rdev(mdk_rdev_t *rdev) { …
 if (bdev == NULL)
    return -ENOMEM;
…}

// Returns NULL if error and sets global
// variable errno
FILE *fopen(const char  path, const char *mode);
```

- Only solves problem of indicating what error occurred
  - First example above also requires many “holes” in the return type that can be used for status codes

# Throwing an Exception in Java

- *Exceptions*: language mechanism for error handling
  - Part of Java, C++, and basically all modern languages

- Upon encountering error, exception is *thrown*
  - Any instance of a subclass of `Exception` can be thrown

```
if (i ≥ 0 && i < a.length)
  return a[i];
throw new ArrayIndexOutOfBounds();//*
```

* *Java exceptions have icky names*

# Catching Exceptions

```
try {
  if (i==0) return;
  throw new ArrayIndexOutOfBounds();
}
catch (ArrayIndexOutOfBounds e) {
  // e is bound to the exception object
  System.out.println(“a[] out of bounds”);
}
```

- In `try { stmts; } catch (Exn e) { more_stmts }`, either
  - If `stmts` executes normally, `more_stmts` never run
  - `stmts` throws at exception, which jumps to the exception handler and runs `more_stmts`

# Try…Catch…Finally Details

```
try { stmts }
catch (Exn_1 e) { stmts_1; }
catch (Exn_2 e) { stmts_2; }
…
finally { stmts_fin; }
```

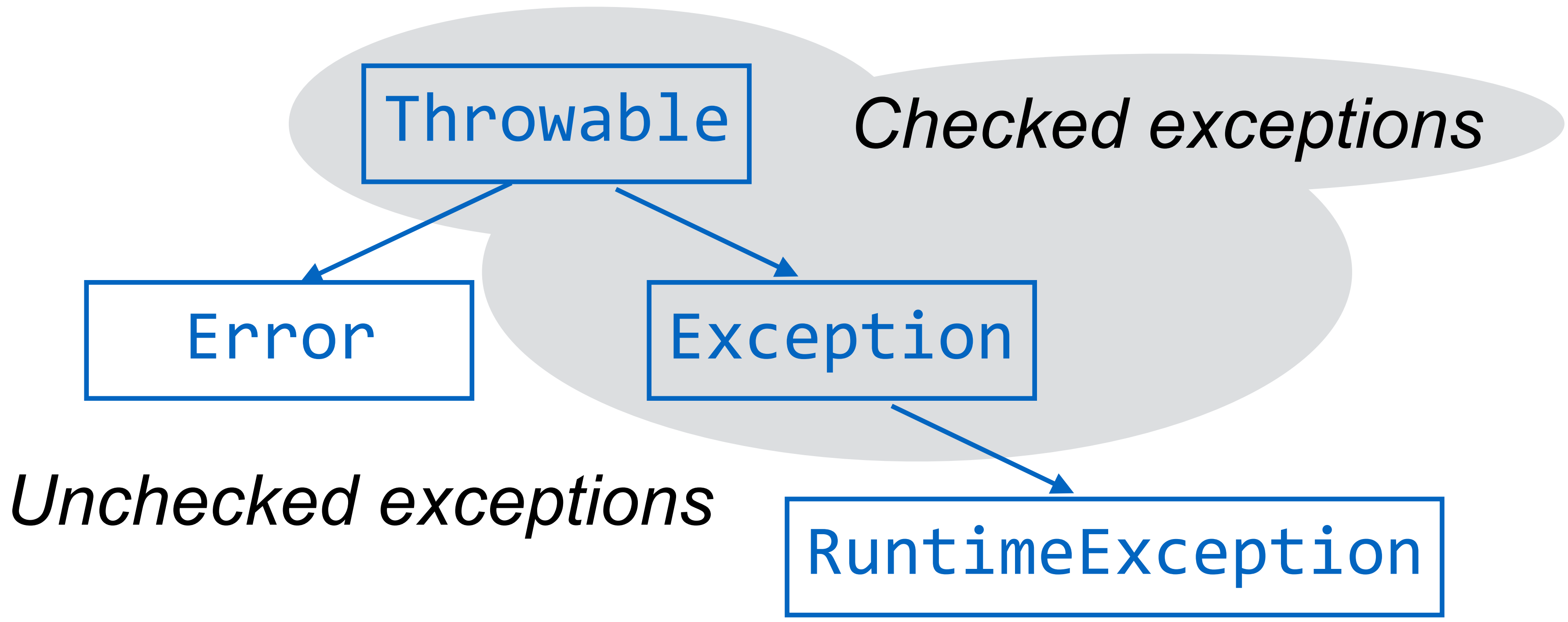
- Caught by first catch with superclass of thrown exn
- If `try…catch` inside of stmts catches exn, it doesn't reach outer catch unless it's rethrown
- `finally` clause is *always* executed before the `try..catch…finally` finishes executing
  - Even if none of the `Exn_i`'s match the thrown exn, or if one of the `stmts_i` throws an exception again
  - Mostly useful for "cleanup" code that must run

# Uncaught Exceptions

- If exception not caught by  program, JVM catches it
  - JVM halts the program, prints exception, gives *atack trace*
  - Stack trace = list of methods on the stack, starting from the exception and going up to `main`

- For most programs for this course, most of the time, exceptions are not caught
  - But for real software systems, crashes are not good
  - Try to catch “typical” exceptions and recover

# Exception Hierarchy




- Checked exceptions must be declared by methods that might throw them (including transitively)

```
public void openNext() throws UnknownHostException,
                              EmptyStackException {
…}
```

# Checked vs. Unchecked Exns

- Subclasses of `Error` and `Runtime` exception need not be listed in method specifications
  - `NullPointerException`
  - `IndexOutOfBoundsException`
  - `VirtualMachineError`

- Are checked exceptions a good design?

# Exceptions Can Carry Values

- Exceptions can also carry values
  - Create a new subclass of `Exception` and add some fields and an appropriate constructor

```
class ParseError extends Exception {
  String file; int line, col;
  …
}

throw new ParseError(f, l, c);
```

# Masking Exceptions

- Handle exception and continue

```
while ((s = …) != null) {
  try {
    FileInputStream f = new FileInputStream(s);
    …
  }
  catch (FileNotFoundException e) {
    System.out.println(s + “ not found”);
} }
```

  - Would probably print error to a log file if this were a production system

# Reflecting Exceptions

- Pass exception up to a higher level
  - Recall no need to do anything special to propagate the same exception
  - But occasionally need to change exception so it makes sense in context of API

```
public static int min(int[] a) {
  int m;
  try { m = a[0]; }
  catch (IndexOutOfBoundsException e) {
    throw new EmptyException();
} }
```

  - Here, "`a` is empty" (a new exception would definition is omitted) is more sensible than an index out of bounds error, since the caller doesn't know the implementation of `min`

# Exception Chaining

- Sometimes, tack a new exception onto previous one

```
public static int min(int[] a) {
  int m;
  try { m = a[0]; }
  catch (IndexOutOfBoundsException e) {
    throw new EmptyException(“min”, e);
} }
```

  - Now (assuming we’ve modified the exception type), the empty exception carries the previous exception so we can see more detail if needed for debugging

# Overloading

- You are already with *operator overloading*

```
3 + 3          // integer addition
3.14 + 3.14    // floating point addition
```

  - These are different instructions on the CPU!
  - Instruction depends on types of the operands
- Java allows methods to be overloaded
  - Different methods with same name declared in class
  - Exact method chosen depends on argument types
  - Choice of method is *compile time* decision
    - Does not involve dynamic dispatch

# Java Overloading Example

```
class Parent {
  void m(int x) { System.out.println(“1”); }
  void m(Object s) { System.out.println(“2”); }
}
class Child extends Parent {
  void m(String s) { System.out.println(“3”); }
}

(new Parent()).m(42);                // prints “1”
(new Parent()).m(new Object()); // prints “2”
(new Parent()).m(“42”);            // prints “2”
(new Child()).m(42);                 // prints “1”
(new Child()).m(new Object());  // prints “2”
(new Child()).m(“42”);             // prints “3”
(new Child()).m((Object) “42”); // prints “2” !
```

- Method selected based on most precise type

# Don’t Use Tricky Overloading

```
class A {
  void m(Object o, String s) {
    System.out.println(“1”);
  }
  void m(String s, Object o) {
    System.out.println(“2”);
  }
}

(new A()).m(new Object(), “42”);  // prints “1”
(new A()).m(“42”, new Object());  // prints “2”
(new A()).m(“42”, “43”);  // error: reference to m
                          // is ambiguous
```