M1:

- Unit review p.449

We can *sequence* commands inside expressions using ';'

- unlike in C, Java, etc., ';' doesn't terminate a statement it *separates* a command from an expression

```
let f (x:int) : int =
  print_string "f called with ";
  print_string (string_of_int x);
  x + x
```

do *not* use ';' here!

note the use of ';' here

The distinction between commands & expressions is artificial.

- `print_string` is a function of type: `string -> unit`
- Commands are actually just expressions of type: `unit`

- Check exams
-

M2:

ResArray (review)

```
public class ResArray {
  …
  private int[] data = {};

  /** Modify the array at position i to contain the value v. */
  public void set(int idx, int val) {
    if (idx >= data.length) {
      int[] newdata = new int[Math.max(idx+1, data.length*2)]
      for (int i=0; i < data.length; i++) {
        newdata[i] = data[i];
      }
      data = newdata;
    }
    data[idx] = val;
  }

  public int[] values() {
    return data;
  }
}
```

Java Abstract Stack Machine

- Similar to OCaml Abstract Stack Machine
  - Workspace
    - Contains the currently executing code
  - Stack
    - Remembers the values of local variables and "what to do next" after function/method calls
  - Heap
    - Stores reference types: objects and arrays
- Key differences:
  - Everything, including stack slots, is mutable by default
  - Objects store *what class was used to create them*
  - *Arrays store type information and length*
  - *New component: Class table (coming soon)*

Heap Reference Values

Arrays

- Type of values that it stores
- Length
- Values for all of the array elements

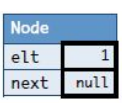
```
int [] a =
    { 0, 0, 7, 0 };
```

length *never* mutable; elements *always* mutable

Objects

- Name of the class that constructed it
- Values for all of the fields

```
class Node {
    private int elt;
    private Node next;
    …
}
```



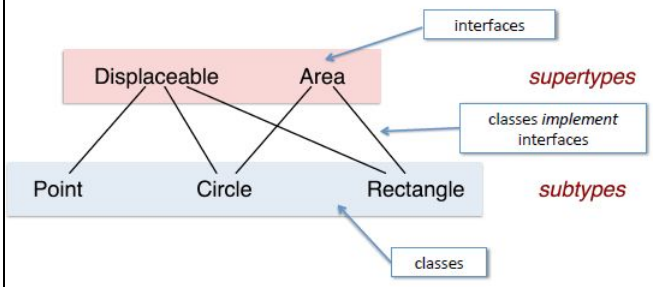
fields may or may not be mutable public/private not tracked by ASM

Object encapsulation

- ***All modification to the state of the object must be done using the object's own methods.***
- Use encapsulation to preserve invariants about the state of the object.
- Enforce encapsulation by not returning aliases from methods.

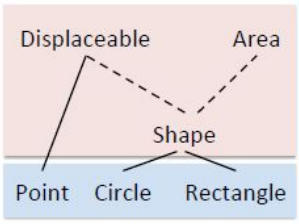
Subtypes and Supertypes

- An interface represents a *point of view* about an object
- Classes can implement *multiple* interfaces



Types can have many different supertypes / subtypes

Interface Hierarchy




```
class Point implements Displaceable {
  … // omitted
}
class Circle implements Shape {
  … // omitted
}
class Rectangle implements Shape {
  … // omitted
}
```

- Shape is a *subtype* of both Displaceable and Area.
- Circle and Rectangle are both subtypes of Shape, and, by *transitivity*, both are also subtypes of Displaceable and Area.
- Note that one interface may extend *several* others.
  - Interfaces do not necessarily form a tree, but the hierarchy has no cycles.

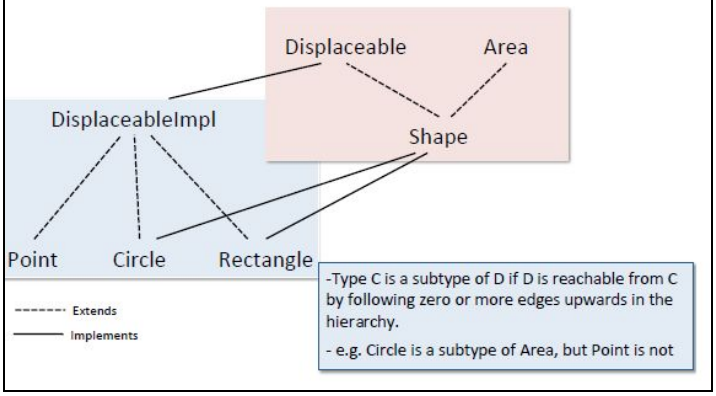
Class Extension: Inheritance

- Classes, like interfaces, can also extend one another.
  - Unlike interfaces, a class can extend only *one* other class.
- The extending class *inherits* all of the fields and methods of its *superclass*, may include additional fields or methods.
  - This captures the "is a" relationship between objects (e.g. a Car is a Vehicle).
  - Class extension should *never* be used when "is a" does not relate the subtype to the supertype.

Simple Inheritance

- In *simple inheritance*, the subclass only *adds* new fields or methods.
- Use simple inheritance to *share common code* among related classes.
- Example: Circle, and Rectangle have *identical* code for getX(), getY(), and move() methods when implementing Displaceable.

Subtyping with Inheritance



-Type C is a subtype of D if D is reachable from C by following zero or more edges upwards in the hierarchy.
- e.g. Circle is a subtype of Area, but Point is not

Subtype Polymorphism*

- Main idea:

  Anywhere an object of type A is needed, an object that is a subtype of A can be provided.

```
// in class C
public static void leapIt(DisplaceableImpl c) {
   c.move(1000,1000);
 }
// somewhere else
C.leapIt(new Circle (10));
```

- If B is a subtype of A, it provides all of A's (public) methods.
- Due to dynamic dispatch, the behavior of a method depends on B's implementation.
  - Simple inheritance means B's method is inherited from A
  - Otherwise, behavior of B should be "compatible" with A's behavior

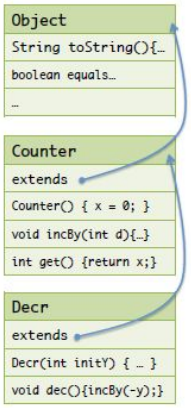
*polymorphism = many shapes

- L25

ASM refinement: The Class Table

```
public class Counter {
  private int x;
  public Counter () { x = 0; }
  public void incBy(int d) { x = x + d; }
  public int get() { return x; }
}

public class Decr extends Counter {
  private int y;
  public Decr (int initY) { y = initY; }
  public void dec() { incBy(-y); }
}
```

Class Table

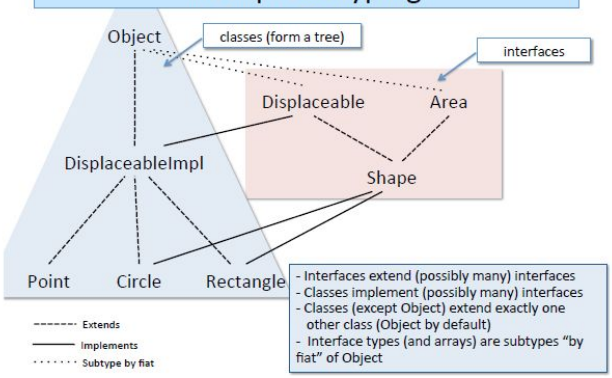


The class table contains:

- the code for each method,
- references to each class's parent, and
- the class's static members.

- **L26 Java Generics (L26,7)**

Recap: Subtyping



- Interfaces extend (possibly many) interfaces
- Classes implement (possibly many) interfaces
- Classes (except Object) extend exactly one other class (Object by default)
- Interface types (and arrays) are subtypes "by fiat" of Object

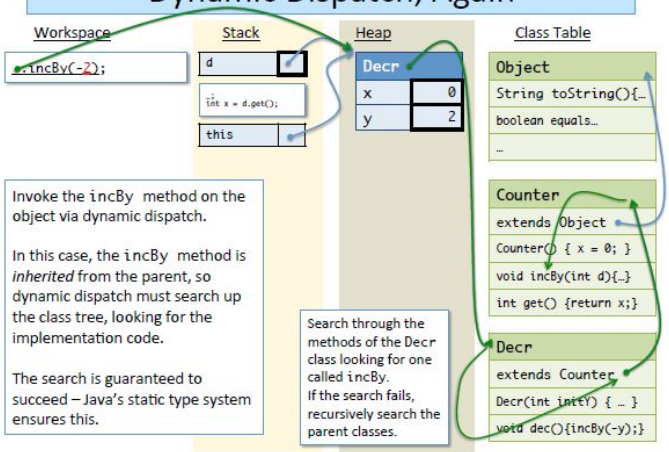
Summary: `this` and dynamic dispatch

- When object's method is invoked, as in `o.m()`, the code that runs is determined by `o`'s *dynamic* class.
  - The dynamic class, represented as a pointer into the class table, is included in the object structure in the heap
  - If the method is inherited from a superclass, determining the code for `m` might require searching up the class hierarchy via pointers in the class table
  - This process of *dynamic dispatch* is the heart of OOP!
- Once the code for m has been determined, a binding for `this` is pushed onto the stack.
  - The `this` pointer is used to resolve field accesses and method invocations inside the code.

Refinements to the Stack Machine

- Code is stored in a *class table*, which is a special part of the heap:
  - When a program starts, the JVM initializes the class table
  - Each class has a pointer to its (unique) parent in the class tree
  - A class stores the constructor and method code for its instances
  - The class also stores *static* members
- Constructors:
  - Allocate space in the heap
  - (Implicitly) invoke the superclass constructor, then run the constructor body
- Objects and their methods:
  - Each object in the heap has a pointer to the class table of its dynamic type (the one it was created with via `new`).
  - A method invocation "`o.m(…)`" uses `o`'s class table to "dispatch" to the appropriate method code (might involve searching up the class hierarchy).
  - Methods and constructors take an implicit "`this`" parameter, which is a pointer to the object whose method was invoked. Fields& methods are accessed with `this.`

Dynamic Dispatch, Again



Invoke the `incBy` method on the object via dynamic dispatch.

In this case, the `incBy` method is *inherited* from the parent, so dynamic dispatch must search up the class tree, looking for the implementation code.

The search is guaranteed to succeed – Java's static type system ensures this.

Search through the methods of the `Decr` class looking for one called `incBy`.
If the search fails, recursively search the parent classes.

Static Members

- Classes in Java can also act as *containers* for code and data.
- The modifier `static` means that the field or method is associated with the class and *not* instances of the class.

```
class C {
  public static int x = 23;
  public static int someMethod(int y) { return C.x + y; }
  public static void main(String args[]) {
    …
  }
}

 // Elsewhere:
C.x = C.x + 1;
C.someMethod(17);
```

You can do a static assignment to initialize a static field.

Access to the static member uses the class name `C.x` or `C.foo()`

Class Table Associated with C

- The class table entry for C has a field slot for x.
- Updates to `C.x` modify the contents of this slot: `C.x = 17;`

C	
extends Object	
static x	23
static int someMethod(int y) { return x + y; }	
static void main(String args[]) {…}	

- A static field is a *global* variable
  - There is only one heap location for it (in the class table)
  - Modifications to such a field are visible everywhere the field is
    - if the field is public, this means *everywhere*
  - Use with care!

Static Methods (Details)

- Static methods do *not* have access to a `this` pointer
  - Why? There isn't an instance to dispatch through!
  - Therefore, static methods may only directly call other static methods.
  - Similarly, static methods can only directly read/write static fields.
  - Of course a static method can create instance of objects (via `new`) and then invoke methods on those objects.

- Gotcha: It is possible (but confusing) to invoke a static method as though it belongs to an object instance.
  - e.g. `o.someMethod(17)` where `someMethod` is static
  - Eclipse will issue a warning if you try to do this.

Parametric Polymorphism (a.k.a. Generics)

- Big idea:

  Parameterize a type (i.e. interface or class) by another type.

```
public interface Queue<E> {
  public void enq(E o);
  public E deq();
  public boolean isEmpty();
}
```

- The implementation of a parametric polymorphic interface cannot depend on the implementation details of the parameter.
  - e.g. the implementation of `enq` cannot invoke any methods on 'o'

M3:

- L27 -Java Generics - Collections and Equality

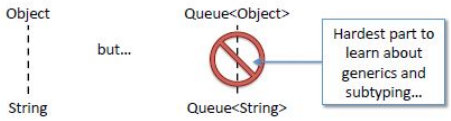
Subtyping and Generics*

```
Queue<String> qs = new QueueImpl<String>();
Queue<Object> qo = qs;

qo.enq(new Object());
String s = qs.deq();
```

Ok? Sure!
Ok? Let's see…

Ok? I guess
Ok? NOOOO!

- Java generics are *invariant*:
  - Subtyping of *arguments* to generic types does not imply subtyping between the instantiations:



* Subtyping and generics interact in other ways too. Java supports *bounded polymorphism* and *wildcard types*, but those are beyond the scope of CIS 120.

Subtype Polymorphism

```
interface ObjQueue {
   public void enq(Object elt);
   public Object deq();
   public boolean isEmpty();
}
```

```
ObjQueue q = …;

q.enq(" CIS 120 ");
Object  x = q.deq();
//System.out.println(x.trim());
q.enq(new Point(0.0,0.0));
___B___  y = q.deq();
```

What type for B?

1. Point
2. Object
3. ObjQueue
4. None of the above

ANSWER: Object

Subtype Polymorphism

```
interface ObjQueue {
   public void enq(Object elt);
   public Object deq();
   public boolean isEmpty();
}
```

```
ObjQueue q = …;

q.enq(" CIS 120 ");
Object  x = q.deq();
System.out.println(x.trim());
```

← Does this line type check

1. Yes
2. No
3. It depends

ANSWER: No

Generics (Parametric Polymorphism)

```
public interface Queue<E> {
  public void enq(E o);
  public E deq();
  public boolean isEmpty();
  …
}
```

```
Queue<String> q = …;

q.enq(" CIS 120 ");
String x = q.deq();              // What type of x?   String
System.out.println(x.trim());    // Is this valid?    Yes!
q.enq(new Point(0.0,0.0));       // Is this valid?    No!
```

Subtyping and Generics

Which of these are true, assuming that class QueueImpl<E> implements interface Queue<E>?

1. QueueImpl<Queue<String>> is a subtype of Queue<Queue<String>>
2. Queue<QueueImpl<String>> is a subtype of Queue<Queue<String>>
3. Both
4. Neither

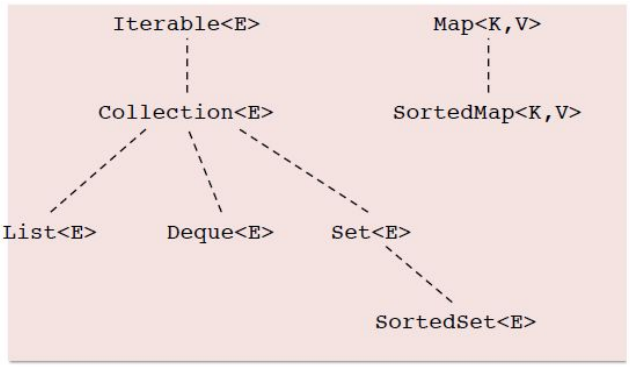
Answer: 1

One Subtlety

- Unlike OCaml, Java classes and methods can be generic only with respect to *reference* types.
  - Not possible to do: `Queue<int>`
  - Must instead do: `Queue<Integer>`
- Java Arrays cannot be generic:
  - Not possible to do:

```
class C<E> {
   E[] genericArray;
   public C() {
      genericArray = new E[];
   }
}
```

Interfaces* of the Collections Library



*not all of them!

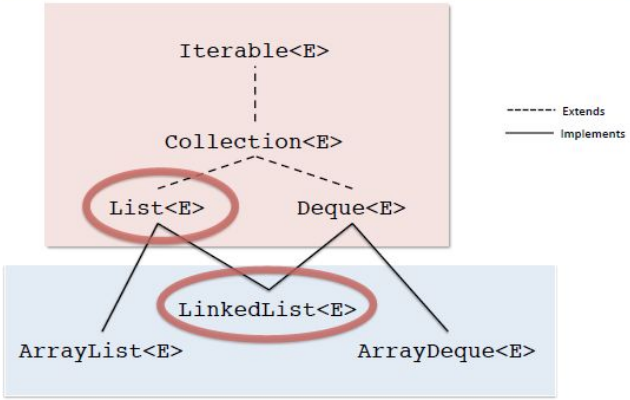
Collection<E> Interface (Excerpt)

```
interface Collection<E> extends Iterable<E> {
  // basic operations
  int size();
  boolean isEmpty();
  boolean add(E o);
  boolean remove(Object o);     // why not E?*
  boolean contains(Object o);

  // bulk operations
  ...
}
```

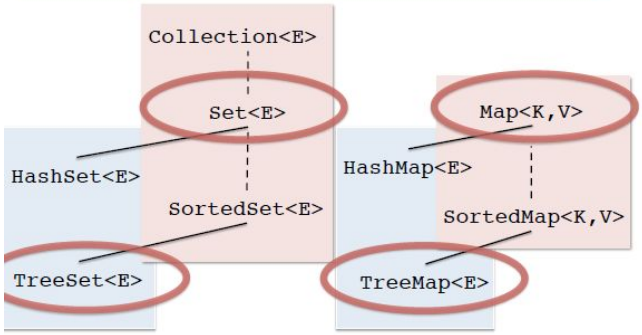
- We've already seen this interface in the OCaml part of the course.
- Most collections are designed to be *mutable* (like queues)

* Why not E? Internally, collections use the `equals` method to check for equality – membership is determined by o.equals, which does not have to be false for objects of different types. Most applications only store and remove one type of element in a collection, in which case this subtlety never becomes an issue.

Sequences



Sets and Maps*



*Read javadocs before instantiating these classes! There are some important details to be aware of to use them correctly.

Adding Comparable to Point

```
import java.util.*;

class Point implements Comparable<Point> {
    private final int x, y;
    public Point(int x0, int y0) { x = x0; y = y0; }
    public int getX(){ return x; }
    public int getY(){ return y; }
    @Override
    public int compareTo(Point o) {
        if (this.x < o.x) {
            return -1;
        } else if (this.x > o.x) {
            return 1;
        } else if (this.y < o.y) {
            return -1;
        } else if (this.y > o.y) {
            return 1;
        }
        return 0;
    }
}
```

A Subclass can *Override* its Parent

```
class C {
      public void printName() { System.out.println("I'm a C"); }
}

class D extends C {
      public void printName() { System.out.println("I'm a D"); }
}

// somewhere in main
C c = new D();
c.printName();
```

- Our ASM model for dynamic dispatch already explains what will happen when we run this code.
- Useful for changing the default behavior of classes.
- But… can be confusing and difficult to reason about if not used carefully.

Difficulty with Overriding

```
class C {

  public void printName() {
    System.out.println("I'm a " + getName());
  }

  public String getName() {
    return "C";
  }
}

class E extends C {

  public String getName() {
    return "E";
  }
}

// in main
C c = new E();
c.printName();
```

The C class might be in another package, or a library…

Whoever wrote E might not be aware of the implications of changing `getName`.

Overriding the method causes the behavior of `printName` to change!

- Overriding can break invariants/ abstractions relied upon by the superclass.

When to override equals

- In classes that represent immutable *values*
  - String already overrides equals
  - Our Point class is a good candidate
- When there is a "logical" notion of equality
  - The collections library overrides equality for Sets (e.g. two sets are equal if and only if they contain equal elements)
- Whenever instances of a class might need to serve as *elements of a set* or as *keys in a map*
  - The collections library uses `equals` internally to define set membership and key lookup
  - (This is the problem with the example code)

When *not* to override equals

- When each instance of a class is inherently unique
  - *Often* the case for mutable objects (since its state might change, the only sensible notion of equality is identity)
  - Classes that represent "active" entities rather than data (e.g. threads, gui components, etc.)
- When a superclass already overrides equals and provides the correct functionality.
  - Usually the case when a subclass is implemented by adding only new methods, but not fields

The contract for equals

- The equals method implements an *equivalence relation* on non-null objects.
- It is *reflexive*:
  - for any non-null reference value x, x.equals(x) should return true
- It is *symmetric*:
  - for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true
- It is *transitive*:
  - for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
- It is consistent:
  - for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the object is modified
- For any non-null reference x, x.equals(null) should return false.

Directly from: http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#equals(java.lang.Object)

`instanceof`

- The `instanceof` operator tests the *dynamic* type of any object

```
Point p = new Point(1,2);
Object o1 = p;
Object o2 = "hello";
System.out.println(p instanceof Point);
       // prints true
System.out.println(o1 instanceof Point);
       // prints true
System.out.println(o2 instanceof Point);
       // prints false
System.out.println(p instanceof Object);
       // prints true
```

- But… use `instanceof` judiciously – usually dynamic dispatch is better.

Refining the equals implementation

```
@Override
public boolean equals(Object o) {
    boolean result = false;
    if (o instanceof Point) {
        Point that = (Point) o;
        result = (this.getX() == that.getX() &&
                  this.getY() == that.getY());
    }
    return result;
}
```

This cast is guaranteed to succeed.

Suppose we extend Point like this

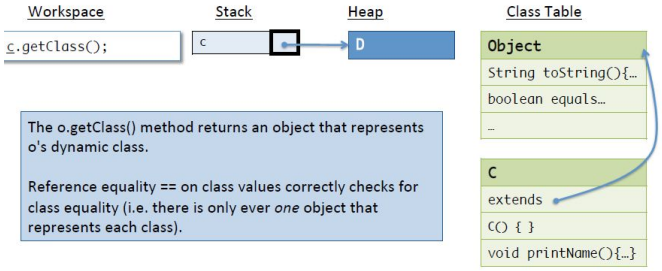
```
public class ColoredPoint extends Point {
  private final int color;
  public ColoredPoint(int x, int y, int color) {
    super(x,y);
    this.color = color;
  }

  @Override
  public boolean equals(Object o) {
    boolean result = false;
    if (o instanceof ColoredPoint) {
      ColoredPoint that = (ColoredPoint) o;
      result = (this.color == that.color &&
                super.equals(that));
    }
    return result;
  }
}
```

This version of equals is suitably modified to check the color field too.

Keyword super is used to invoke overridden methods.

Should equality use instanceof?

- To correctly account for subtyping, we need the classes of the two objects to match *exactly*.
- instanceof only lets us ask about the subtype relation
- How do we access the dynamic class?



The o.getClass() method returns an object that represents o's dynamic class.

Reference equality == on class values correctly checks for class equality (i.e. there is only ever *one* object that represents each class).

Correct Implementation: Point

```
@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Point other = (Point) obj;
    if (x != other.x)
        return false;
    if (y != other.y)
        return false;
    return true;
}
```

Check whether obj is a Point.

break

- GOTCHA: By default, each branch will "fall through" into the next, sot that code prints:

```
Got CREATE!
Got MESG!
Got NICK!
default
```

- Use an explicit break to avoid fallthrough:

```
switch (t) {
case CREATE : System.out.println("Got CREATE!");
              break;
case MESG   : System.out.println("Got MESG!");
              break;
case NICK   : System.out.println("Got NICK!");
              break;
default: System.out.println("default");
}
```

Equality and Hashing

- Whenever you override equals you ***must also*** override hashCode in a compatible way
  - If `o1.equals(o2)` then `o1.hashCode() == o2.hashCode()`
  - hashCode is used by the HashSet and HashMap collections
- Forgetting to do this can lead to extremely puzzling bugs!

Enumerations

Enum types are just a convenient way of defining a class along with some standard methods.

- Enum types (implicitly) extend java.lang.Enum
- They can contain constant data "properties"
- As classes, they can have methods -- e.g. to access a field
- Intended to represent constant data values

Automatically generated static methods:

- `valueOf` : converts a `String` to an Enum

  Command.Type c = Command.Type.valueOf ("CONNECT");
- `values`: returns an `Array` of all the enumerated constants

  Command.Type[] varr = Command.Type.values();

Iterator and Iterable

```
interface Iterator<E> {
    public boolean hasNext();
    public E next();
    public void delete();   // optional
}
```

```
interface Iterable<E> {
    public Iterator<E> iterator();
}
```

- L29 - Enums, Iterators, Exceptions / Ch27 (69)

syntax:

```
// repeat body until condition becomes false
while (condition) {
  body
}
```

statement

boolean *guard* expression

example:

```
List<Book> shelf = …  // create a list of Books

// iterate through the elements on the shelf
Iterator<Book> iter = shelf.iterator();
while (iter.hasNext()) {
    Book book = iter.next();
    catalogue.addInfo(book);
    numBooks = numbooks+1;
}
```

syntax:

```
for (init-stmt; condition; next-stmt) {
  body
}
```

equivalent while loop:

```
init-stmt;
while (condition) {
  body
  next-stmt;
}
```

```
List<Book> shelf = …  // create a list of Books

// iterate through the elements on the shelf
for (Iterator<Book> iter = shelf.iterator();
     iter.hasNext();) {
   Book book = iter.next();
   catalogue.addInfo(book);
   numBooks = numbooks+1;
}
```

For-each Loops

syntax:

```
// repeat body for each element in collection
for (type var : coll) {
  body
}
```

element type E

Array of E or instance of Iterable<E>

example:

```
List<Book> shelf = …  // create a list of books

// iterate through the elements on a shelf
for (Book book : shelf) {
    catalogue.addInfo(book);
    numBooks = numbooks+1;
}
```

For-each Loops (cont'd)

Another example:

```
int[] arr = …  // create an array of ints

// count the non-null elements of an array
for (int elt : arr) {
    if (elt != 0) cnt = cnt+1;
}
```

For-each can be used to iterate over arrays or any class that implements the `Iterable<E>` interface (notably `Collection<E>` and its subinterfaces).

Iterator example

```
public static void iteratorExample() {
    List<Integer> nums = new LinkedList<Integer>();
    nums.add(1);
    nums.add(2);
    nums.add(7);

    int numElts = 0;
    int sumElts = 0;
    Iterator<Integer> iter =
        nums.iterator();
    while (iter.hasNext()) {
      Integer v = iter.next();
      sumElts = sumElts + v;
      numElts = numElts + 1;
    }

    System.out.println("sumElts = " + sumElts);
    System.out.println("numElts = " + numElts);
  }
```

What is printed by iteratorExample()?

1. sumElts = 0 numElts = 0
2. sumElts = 3 numElts = 2
3. sumElts = 10 numElts = 3
4. NullPointerException
5. Something else

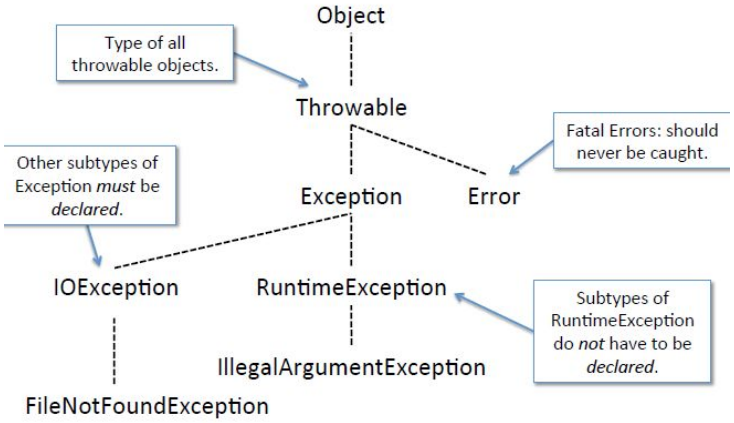
Answer: 3

Exceptions

- An exception is an *object* representing an abnormal condition
  - Its internal state describes what went wrong
  - e.g. NullPointerException, IllegalArgumentException, IOException
  - Can define your own exception classes
- *Throwing* an exception is an *emergency exit* from the current context
  - The exception propagates up the invocation stack until it either reaches the top of the stack, in which case the program aborts with the error, or the exception is *caught*
- *Catching* an exception lets callers take appropriate actions to handle the abnormal circumstances

- L30 -, Exceptions / Ch27 (144)

Exception Class Hierarchy



Declared vs. Undeclared?

- Tradeoffs in the software design process:
- *Declared*: better documentation
  - forces callers to acknowledge that the exception exists
- *Undeclared*: fewer static guarantees (compiler can help less)
  - but, much easier to refactor code
- In practice: test-driven development encourages "fail early/fail often" model of code design and lots of code refactoring, so "undeclared" exceptions are prevalent.

- A reasonable compromise:
  - Use declared exceptions for libraries, where the documentation and usage enforcement are critical
  - Use undeclared exceptions in client code to facilitate more flexible development

Checked (Declared) Exceptions

- Exceptions that are subtypes of `Exception` but not `RuntimeException` are called *checked* or *declared*.
- A method that might throw a checked exception must declare it using a "throws" clause in the method type.
- The method might raise a checked exception either by:
  - directly throwing such an exception

```
public void maybeDoIt (String file) throws AnException {
       if (…) throw new AnException();  // directly throw
       …
```

  - or by calling another method that might itself throw a checked exception

```
public void doSomeIO (String file) throws IOException {
       Reader r = new FileReader(file);  // might throw
       …
```

Unchecked (Undeclared) Exceptions

- Subclasses of RuntimeException *do not* need to be declared via "throws"
  - even if the method does not explicitly handle them.
- Many "pervasive" types of errors cause RuntimeExceptions
  - NullPointerException
  - IndexOutOfBoundsException
  - IllegalArgumentException

```
public void mightFail (String file) {
       if (file.equals("dictionary.txt") {
          // file could be null!
       …
```

- The original intent was that such exceptions represent disastrous conditions from which it was impossible to sensibly recover…

Checked vs. Unchecked Exceptions

Which methods need a "throws" clause?

Note: IllegalArgumentException is a subtype of RuntimeException.

IOException is not.

1) all of them
2) none of them
3) m and n
4) n only
5) n, r, and s
6) n, q, and s
7) m, p, and s
8) something else

Answer:
n, q and s should say throws IOException

```
public class ExceptionQuiz {
    public void m(Object x) {
      if (x == null)
          throw new IllegalArgumentException();
    }
    public void n(Object y) {
      if (y == null) throw new IOException();
    }
    public void p() {
      m(null);
    }
    public void q() {
      n(null);
    }
    public void r() {
      try { n(null); } catch (IOException e) {}
    }
    public void s() {
      n(new Object());
    }
}
```

- L31- I/O & Histogram Demo / Ch28 (210)

Good Style for Exceptions

- In Java, exceptions should be used to capture *exceptional circumstances*
  - Try/catch/throw incur performance costs and complicate reasoning about the program, don't use them when better solutions exist
- *Re-use existing exception types* when they are meaningful to the situation
  - e.g. use NoSuchElementException when implementing a container
- Define your own subclasses of Exception if doing so can convey useful information to possible callers that can handle the exception.

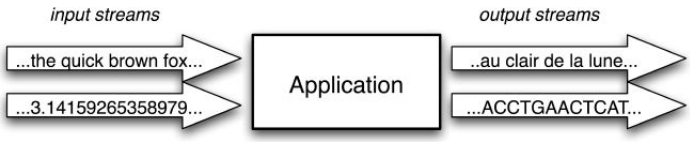
Good Style for Exceptions

- It is often sensible to catch one exception and re-throw a different (more meaningful) kind of exception.
  - e.g. when implementing `WordScanner` (in upcoming lectures), we catch `IOException` and throw `NoSuchElementException` in the `next` method.
- Catch exceptions as near to the source of failure as makes sense
  - i.e. where you have the information to deal with the exception
- Catch exceptions with as much precision as you can

```
BAD:    try {…} catch (Exception e) {…}
BETTER: try {…} catch (IOException e) {…}
```

I/O Streams

- The *stream* abstraction represents a communication channel with the outside world.
  - can be used to read or write a potentially unbounded number of data items (unlike a list)
  - data items are read from or written to a stream one at a time
- The Java I/O library uses subtyping to provide a unified view of disparate data sources and sinks.



InputStream and OutputStream

- Abstract classes that provide basic operations for the Stream class hierarchy:

```
int read ();        // Reads the next byte of data
void write (int b); // Writes the byte b to the output
```

- These operations read and write `int` values that represent *bytes*
  - range `0-255` represents a byte value
  - `-1` represents "no more data" (when returned from read)
- java.io provides many subclasses for various sources/sinks of data:
  - files, audio devices, strings, byte arrays, serialized objects
- Subclasses also provides rich functionality:
  - encoding, buffering, formatting, filtering

Binary IO example

```
InputStream fin = new FileInputStream(filename);

int[] data = new int[width][height];
for (int i=0; i < data.length; i++) {
   for (int j=0; j < data[0].length; j++) {
      int ch = fin.read();
      if (ch == -1) {
        fin.close();
        throw new IOException("File ended early");
      }
      data[j][i] = ch;
   }
}
fin.close();
```

BufferedInputStream

- Reading one byte at a time can be slow!
- Each time a stream is read there is a fixed overhead, plus time proportional to the number of bytes read.

  disk -> operating system -> JVM -> program
  disk -> operating system -> JVM -> program
  disk -> operating system -> JVM -> program

- A `BufferedInputStream` presents the same interface to clients, but internally reads many bytes at once into a *buffer* (incurring the fixed overhead only once)

  disk -> operating system ->>>> JVM -> program
  JVM -> program
  JVM -> program
  JVM -> program

Buffering Example

```
FileInputStream fin1 = new FileInputStream(filename);
InputStream fin = new BufferedInputStream(fin1);

int[] data = new int[width][height];
for (int i=0; i < data.length; i++) {
   for (int j=0; j < data[0].length; j++) {
      int ch = fin.read();
      if (ch == -1) {
        fin.close();
        throw new IOException("File ended early");
      }
      data[j][i] = ch;
   }
}
fin.close();
```

Reader and Writer

- Similar to the `InputStream` and `OutputStream` classes, including:

```
int read ();        // Reads the next character
void write (int b); // Writes the char to the output
```

- These operations read and write `int` values that represent *unicode characters*
  - read returns an integer in the range 0 to 65535 (i.e. 16 bits)
  - value `-1` represents "no more data" (when returned from read)
  - requires an "encoding" (e.g. UTF-8 or UTF-16, set by a `Locale`)
- Like byte streams, the library provides many subclasses of Reader and Writer Subclasses also provides rich functionality.
  - use these for portable text I/O
- Gotcha: `System.in`, `System.out`, `System.err` are *byte* streams
  - So wrap in an InputStreamReader / PrintWriter if you need unicode console I/O

PrintStream Methods

`PrintStream` adds buffering and binary-conversion methods to `OutputStream`

```
void println(boolean b); // write b followed by a new line
void println(String s);  // write s followed by a newline
void println();          // write a newline to the stream

void print(String s);    // write s without terminating the line
                         //   (output may not appear until the stream is flushed)
void flush();            // actually output characters waiting to be sent
```

- Note the use of *overloading*: there are *multiple* methods called `println`
  - The compiler figures out which one you mean based on the number of arguments, and/or the *static* type of the argument you pass in at the method's call site.
  - The java I/O library uses overloading of constructors pervasively to make it easy to "glue together" the right stream processing routines

- L34 - Swing I: Drawing and Event Handling / Ch29 (291)

Terminology overview

	GUI (OCaml)	Swing
Graphics Context	`Gctx.gctx`	`Graphics`
Widget type	`Widget.widget`	`JComponent`

Basic Widgets	`button`
	`label`
	`checkbox`
		`JButton`
		`JLabel`
		`JCheckBox`

Container Widgets	`hpair, vpair`	`JPanel, Layouts`

Events	`event`	`ActionEvent`
		`MouseEvent`
		`KeyEvent`

Event Listener	`mouse_listener`
	`mouseclick_listener`
	(any function of type `event -> unit`)
		`ActionListener`
		`MouseListener`
		`KeyListener`

Swing practicalities

- Java library for GUI development
  - javax.swing.*
- Built on existing library: AWT
  - java.awt.*
  - When there are two versions of something, use Swing's. (e.g., java.awt.Button vs. javax.swing.JButton)
    - The "Jxxx" version is usually the one you want, rather than "xxx".
- Portable
  - Communicates with underlying OS's native window system
  - Same Java program looks appropriately different when run on PC, Linux, and Mac

Recursive function for drawing

```
private static void fractal(Graphics gc, int x, int y,
          double angle, double len) {

   if (len > 1) {
     double af = (angle * Math.PI) / 180.0;
     int nx = x + (int)(len * Math.cos(af));
     int ny = y + (int)(len * Math.sin(af));

     gc.drawLine(x, y, nx, ny);

     fractal(gc, nx, ny, angle + 20, len - 8);
     fractal(gc, nx, ny, angle - 10, len - 8);
   }
}
```

Fundamental class: JComponent

- Analogue to widget type from GUI project
  - (*Terminology*: widget == JComponent)
- Subclasses *override* methods
  - paintComponent (like repaint, displays the component)
  - getPreferredSize (like size, calculates the size of the component)
  - Events handled by listeners (don't need to use overriding...)
- Much more functionality available
  - minimum/maximum size
  - font
  - foreground/background color
  - borders
  - what is visible
  - many more...

Simple Drawing Component

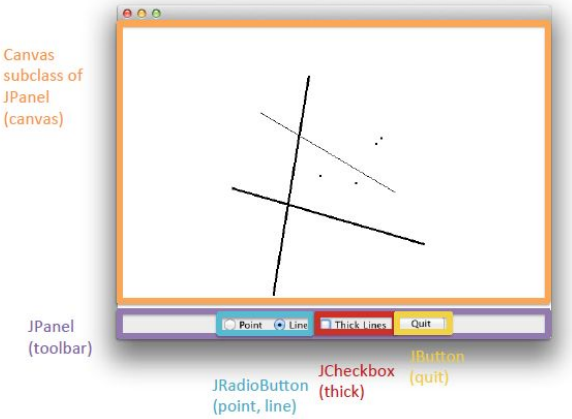
```
public class DrawingCanvas extends JComponent {

    public void paintComponent(Graphics gc) {
        super.paintComponent(gc);

        // set the pen color to green
        gc.setColor(Color.GREEN);

        // draw a fractal tree
        fractal (gc, 200, 450, 270, 80);
    }

    // get the size of the drawing panel
    public Dimension getPreferredSize() {
        return new Dimension(200,200);
    }
```

How to display this component?

JFrame

- Represents a top-level window
  - Displayed directly by OS (looks different on Mac, PC, etc.)
- Contains JComponents
- Can be moved, resized, iconified, closed

```
public void run() {
    JFrame frame = new JFrame("Tree");

    // set the content of the window to be the drawing
    frame.getContentPane().add(new DrawingCanvas());

    // make sure the application exits when the frame closes
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // resize the frame based on the size of the panel
    frame.pack();

    // show the frame
    frame.setVisible(true);
}
```



Inner Classes

- Useful in situations where two objects require "deep access" to each other's internals
- Replaces tangled workarounds like "owner object"
  - Solution with inner classes is easier to read
  - No need to allow public access to instance variables of outer class
- Also called "dynamic nested classes"

Basic Example

Key idea: Classes can be *members* of other classes...

```
class Outer {
  private int outerVar;
  public Outer () {
    outerVar = 6;
  }
  public class Inner {
    private int innerVar;
    public Inner(int z) {
      innerVar = outerVar + z;
    }
    public int getInnerVar() {
      return innerVar;
    }
  }
}
```

Name of this class is Outer.Inner (which is also the static type of objects that this class creates)

Inner class can refer to fields bound in outer class

Constructing Inner Class Objects

Based on your understanding of the Java object model, which of the following make sense as ways to construct an object of an inner class type?

```
1. Outer.Inner obj =  new Outer.Inner()
2. Outer.Inner obj = (new Outer()).new Inner();
3. Outer.Inner obj = new Inner();
4. Outer.Inner obj = Outer.Inner.new ()
```

Answer: 2 – the inner class instances can refer to non-static fields of the outer class (even in the constructor), so the invocation of "new" must be relative to an existing instance of the Outer class.

- **L35 - Swing II: Building GUIs , Inner Classes / Ch29 (314)**

OnOff.java revisited

```
public void run() {
    // ... other code omitted

    LightBulb bulb = new LightBulb();
    panel.add(bulb);

    JButton button = new JButton("On/Off");
    panel.add(button);

    button.addActionListener(new ButtonListener(bulb));

    // ... other code omitted
}
```

Too much boilerplate code...

- ButtonListener really only needs to do bulb.flip()
- But we need all this boilerplate code to build the class
- Often we will only instantiate *one* instance of a given Listener class in a GUI

```
class ButtonListener implements ActionListener {
    private LightBulb bulb;
    public ButtonListener (LightBulb b) {
    bulb = b;
    }
    @Override
    public void actionPerformed(ActionEvent e) {
        bulb.flip();
        bulb.repaint();
    }
}
```

Object Creation

- Inner classes can refer to the instance variables and methods of the outer class
- Inner class instances usually created by the methods/constructors of the outer class

```
public Outer () {
    Inner b = new Inner ();
}
```

Actually this.new

- Inner class instances *cannot* be created independently of a containing class instance.

```
Outer.Inner b = new Outer.Inner()

Outer a = new Outer();
Outer.Inner b = a.new Inner();

Outer.Inner b = (new Outer()).new Inner();
```

Anonymous Inner Classes

- Define a class and create an object from it all at once, inside a method

```
quit.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
       System.exit(0);
   }
});
```

Puts button action right with button definition

```
line.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        shapes.add(new Line(…));
        canvas.repaint();
    }
});
```

Can access fields and methods of outer class, as well as final local variables

Anonymous Inner class

- New *expression* form: define a class and create an object from it all at once

New keyword

```
new InterfaceOrClassName() {
    public void method1(int x) {
        // code for method1
    }
    public void method2(char y) {
        // code for method2
    }
}
```

Normal class definition, no constructors allowed

Static type of the expression is the Interface/superclass used to create it

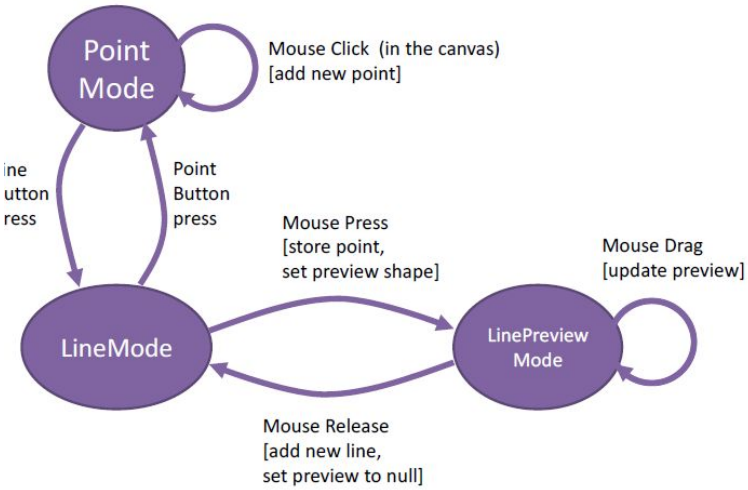
Dynamic class of the created object is anonymous! Can't refer to it.

Like first-class functions

- Anonymous inner classes are roughly* analogous to Ocaml's first-class functions
- Both create "delayed computation" that can be stored in a data structure and run later
  - Code stored by the event / action listener
  - Code only runs when the button is pressed
  - Could run once, many times, or not at all
- Both sorts of computation can refer to variables in the current scope
  - OCaml: Any available variable
  - Java: only instance variables (fields) of enclosing object and local variables marked `final`

- **L36 - Mushroom of Doom/ Ch31 (332)**

Mouse Interaction in Paint



Two interfaces for mouse listeners

```
interface MouseListener extends EventListener {
  public void mouseClicked(MouseEvent e);
  public void mouseEntered(MouseEvent e);
  public void mouseExited(MouseEvent e);
  public void mousePressed(MouseEvent e);
  public void mouseReleased(MouseEvent e);
}
```

```
interface MouseMotionListener extends EventListener {
  public void mouseDragged(MouseEvent e);
  public void mouseMoved(MouseEvent e);
}
```

Updating the Game Sta

```
void tick() {
  if (playing) {
    square.move();
    snitch.move();
    snitch.bounce(snitch.hitWall());       // b
    snitch.bounce(snitch.hitObj(poison)); // .

    if (square.intersects(poison)) {
       playing = false;
       status.setText("You lose!");
    } else if (square.intersects(snitch)) {
       playing = false;
       status.setText("You win!");
    }
    repaint();
  }
}
```

Updating the Game State: keyboard

```
setFocusable(true);
addKeyListener(new KeyAdapter() {
  public void keyPressed(KeyEvent e) {
    if (e.getKeyCode() == KeyEvent.VK_LEFT)
        square.v_x = -SQUARE_VELOCITY;
    else if (e.getKeyCode() == KeyEvent.VK_RIGHT)
        square.v_x = SQUARE_VELOCITY;
    else if (e.getKeyCode() == KeyEvent.VK_DOWN)
        square.v_y = SQUARE_VELOCITY;
    else if (e.getKeyCode() == KeyEvent.VK_UP)
        square.v_y = -SQUARE_VELOCITY;
    }

    public void keyReleased(KeyEvent e) {
        square.v_x = 0;
        square.v_y = 0;
    }
});
```

Make square's velocity nonzero when a key is pressed

Make square's velocity zero when a key is released

MVC Benefits?

- Decouples important "model state" from how that state is presented and manipulated
  - Suggests where to insert interfaces in the design
  - Makes the model testable independent of the GUI
- Multiple views
  - e.g. from two different angles, or for multiple different users
- Multiple controllers
  - e.g. mouse vs. keyboard interaction

MVC Variations

- Many variations on MVC pattern
- Hierarchical / Nested
  - As in the Swing libraries, in which JComponents often have a "model" and a "controller" part
- Coupling between Model / View or View / Controller
  - e.g. in MOD the Model and the View are coupled because the model carries most of the information about the view

// ADVANCED TOPICS

- L37 - Advanced Java Miscellany - Hashing (360)

Hash Sets and Maps: The Big Idea

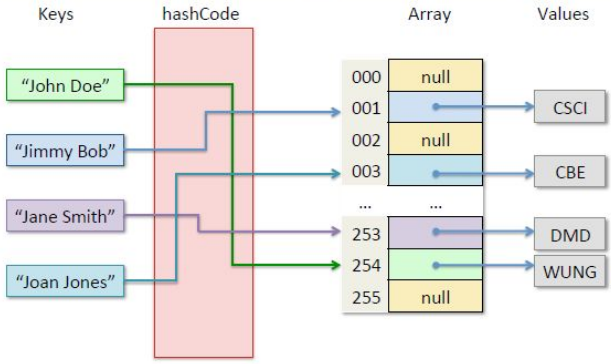
Combine:

- the advantage of arrays:
  - *efficient* random access to its elements
- with the advantage of a map datastructure
  - arbitrary keys (not just integer indices)

How?

- Create an index into an array by *hashing* the data in the key to turn it into an int
  - Java's hashCode method maps key data to ints
  - Generally, the space of keys is much larger than the space of hashes, so, unlike array indices, hashCodes might not be unique

Hash Maps, Pictorially



A schematic HashMap taking Strings (student names) to Undergraduate Majors. The hashCode takes each string name to an integer code, which we then take "mod 256" to get an array index between 0 and 255.
For example, `"John Doe".hashCode()` mod 256 is 254.

Hashing and User-defined Classes

```
public class Point {
    private final int x;
    private final int y;
    public Point(int x, int y) { this.x = x; this.y = y; }
    public int getX() { return x; }
    public int getY() { return y; }
}

// somewhere else…
Map<Point,String> m = new HashMap<Point,String>();
m.put(new Point(1,2), "House");
System.out.println(m.containsKey(new Point(1,2)));
```

What gets printed to the console?

1. true
2. false
3. I have no idea

ANSWER: 2 – hashCode not implemented

HashCode Requirements

Whenever you override `equals` you must also override `hashCode` in a consistent way:

- whenever `o1.equals(o2) == true` you must ensure that `o1.hashCode() == o2.hashCode()`

Why? Because comparing hashes is supposed to be a quick approximation for equality.

- Note: the converse does not have to hold:
  - `o1.hashcode() == o2.hashCode()` does *not* necessarily mean that o1.equals(o2)

Example for Point

```
public class Point {
   @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + x;
        result = prime * result + y;
        return result;
    }
}
```

- Examples:
  - (new Point(1,2)).hashCode() yields 994
  - (new Point(2,1)).hashCode() yields 1024
- Note that equal points (in the sense of `equals`) have the same hashCode
- Why 31? Prime chosen to create more uniform distribution
- Note: Tools (e.g. eclipse) can *generate* this code

Collections Requirements

- All collections invoke `equals` method on elements
  - Defaults to == (reference equality)
  - Override `equals` to create structural equality
  - Should always be an equivalence relation: reflexive, symmetric, transitive
- HashSets/HashMaps also invoke `hashCode` method on elements
  - Override when equals is overridden
  - Should be "compatible" with `equals`
  - Should try to distribute hash codes uniformly
  - Iterators are not guaranteed to follow order of hashCodes
- Ordered collections (`TreeSet`, `TreeMap`) require element type to implement `Comparable` interface
  - Provide `compareTo` method
  - Should implement a *total order*
  - Should be compatible with equals
    - (i.e. `o1.equals(o2)` exactly when `o1.compareTo(o2) == 0`)

Recipe: Computing Hashes

- What is a good recipe for computing hash values for your own classes?
  - intuition: "smear" the data throughout all the bits of the resulting

1. Start with some constant, arbitrary, non-zero int in `result`.
2. For each significant field f of the class (i.e. each field taken into account when computing equals), compute a "sub" hash code c for the field:
   - For boolean fields: `(f ? 1 : 0)`
   - For byte, char, int, short: `(int) f`
   - For long: `(int) (f ^ (f >>> 32))`
   - For references: 0 if the reference is null, otherwise use the `hashCode()` of the field.
3. Accumulate those subhashes into the result by doing (for each field's c):
   `result = prime * result + c;`
4. return `result`

- **L38 - Advanced Java Miscellany - Concurrency, GC & Memory Management (391)**

Threads

- Java programs can be *multithreaded*
  - more than one "thread" of control operating simultaneously
- A `Thread` object can be created from any class that implements the `Runnable` interface
  - `start`: launch the thread
  - `join`: wait for the thread to finish
- Abstract Stack Machine:
  - Each thread has its own workspace and stack
  - All threads *share* a common heap
  - Threads can communicate via shared references

Uses + Perils

- Threads are useful when one program needs to do multiple things simultaneously:
  - game animation + user input
  - chat server interacting with multiple chat clients
  - hide latency: do work in one thread while another thread waits (e.g. for disk or network I/O)
- Problem: Race Conditions
  - What happens when one thread tries to read a memory location at the same time another thread is writing it?
  - What if more than one thread tries to write different values at the same time?

```
interface Counter {
  public void inc();
  public int get();
}

class UCounter implements Counter {
  private int cnt = 0;

  public void inc() {
    cnt = cnt + 1;
  }

  public int get() {
    return cnt;
  }
}
```

```
// The computation thread simply increments
// the provided counter 1000 times
class CounterUser implements Runnable {
  private Counter c;
  private int id;

  CounterUser(int id, Counter c) {
    this.id = id;
    this.c = c;
  }

  @Override
  public void run() {
    for (int i = 0; i < 1000; i++) {
      // System.out.println("Thread: " + id);
      c.inc();
    }
  }
}
```

```
public class MultiThreaded {

  public static void main(String[] args) {
    Counter c = new UCounter();

    // set up a race on the shared counter c
    Thread t1 = new Thread(new CounterUser(1, c));
    Thread t2 = new Thread(new CounterUser(2, c));
    t1.start();
    t2.start();
    try {
      t1.join();
      t2.join();
    } catch (InterruptedException e) {
    }
    System.out.println("Counter value = " + c.get());
  }

}
```

Create thread 1

Create thread 2

Start thread 1

Start thread 2

Wait for thread 1 to finish

Wait for thread 2 to finish

Asynchronous threading -

Answer: The program with print "Counter value = *val*"
for 1000 >= val >= 2000.
The answer will likely be *different* each time the program is run!!!!

The synchronized keyword

Synchronized methods are *atomic*

- They run without any other threads running

Second Try: use Synchronization

```
//This class uses synchronization
class SynchronizedCounter implements Counter {
  private int cnt = 0;

  public synchronized void inc() {
   cnt = cnt + 1;
  }

  public synchronized int get() {
   return cnt;
  }
}
```

Using The New Counters

```
public class MultiThreaded {

  public static void main(String[] args) {

    Counter c = new SynchronizedCounter();

    // set up a race on the shared counter c
    Thread t1 = new Thread(new CounterUser(1, c));
    Thread t2 = new Thread(new CounterUser(2, c));
    t1.start();
    t2.start();
    try {
      t1.join();
      t2.join();
    } catch (InterruptedException e) {
    }

    System.out.println("Counter value = " + c.get());
  }

}
```

New!!

CIS120

Other Synchronization in Java

Need *thread safe* libraries:

- `java.util.concurrent` has `BlockingQueue` and `ConcurrentMap`
- help rule out synchronization errors
- Note: Swing is *not* thread safe!

- Java also provides *locks*
  - objects that act as synchronizers for blocks of code

- *Deadlock*: cyclic dependency in synchronization of locks
  - Thread A waiting for lock held by B,
    Thread B waiting for lock held by A

Immutability!

- Note that read-only datastructures are immune to race conditions
  - It's OK for multiple threads to read a heap location simultaneously
  - Less need for locking, synchronization

- As always: immutable data structures simplify your code

Real-world example:

FaceBook's Haxl Library

- Library written in Haskell
- Concurrency / Distributed Database
- https://github.com/facebook/Haxl

Why Garbage Collection?

- Manual memory management is cumbersome & error prone:
  - Freeing the same reference twice is ill defined (crashes or other bugs)
  - Explicit `free` isn't modular: To properly free all allocated memory, the programmer has to know what code "owns" each object. Owner code must ensure `free` is called just once.
  - Not calling `free` leads to *space leaks*: memory never reclaimed
    - Many examples of space leaks in long-running programs
- Garbage collection:
  - Have the language runtime system determine when an allocated chunk of memory will no longer be used and free it automatically.
  - Extremely convenient and safe
  - Garbage collection does impose costs (performance, predictability)

Mark and Sweep Garbage Collection

- Classic algorithm with two phases:
- Phase 1: Mark
  - Start from the roots
  - Do depth-first traversal, marking every object reached.
- Phase 2: Sweep
  - Walk over *all* allocated objects and check for marks.
  - Unmarked objects are reclaimed.
  - Marked objects have their marks cleared.
  - Optional: compact all live objects in heap by moving them adjacent to one another. (Needs extra work & indirection to "patch up" references)

Copying Garbage Collection

- Like mark & sweep: collects all garbage.
- Basic idea: use *two* regions of memory
  - One region is the memory in use by the program. New allocation happens in this region.
  - Other region is idle until the GC requires it.
- Garbage collection algorithm:
  - Traverse over live objects in the active region (called the "*from-space*"), copying them to the idle region (called the "*to-space*").
  - After copying all reachable data, switch the roles of the from-space and to-space.
  - All dead objects in the (old) from-space are discarded en masse.
  - A side effect of copying is that all live objects are compacted together.

Memory Use & Reachability

When is a chunk of memory no longer needed?

- In general, this problem is undecidable.

We can approximate this information by freeing memory that can't be reached from any *root* references.

- A *root reference* is one that might be accessible directly from the program (i.e. they're not in the heap).
- Root references include (global) static fields and references in the stack.

If an object can be reached by traversing pointers from a root, it is *live*.

It is safe to reclaim all heap allocations not reachable from a root (such objects are *garbage* or *dead* objects).

Costs & Implications

Need to generalize to account for objects that have multiple outgoing pointers.

Mark & Sweep algorithm reads all memory in use by the program (even if it's garbage!)

- Running time is proportional to the total amount of allocated memory (both live and garbage).
- Can pause the programs for long times during garbage collection.

Garbage Collection Take Aways

Big idea: the Java runtime system tries to free-up as much memory as it can automatically.

- Almost always a big win, in terms of convenience and reliability

Sometimes can affect performance:

- Lots of dead objects might take a long time to collect
- When garbage collection will be triggered can be hard to predict, so there can be "pauses" (modern GC implementations try to avoid this!)
- Global data structures can have references to "zombie" objects that won't be used, but are still reachable ⇒ "space leak".

There are many advanced programming techniques to address these issues:

- Configuring the GC parameters
- Explicitly triggering a GC phase
- "Weak" references

- **L39 - Advanced Java Miscellany - Java 1.8 Functional Programming and Lambdas (430)**

Lambdas – Why and What?

- Helps create instances of single-method classes more easily
- Think of them as anonymous methods

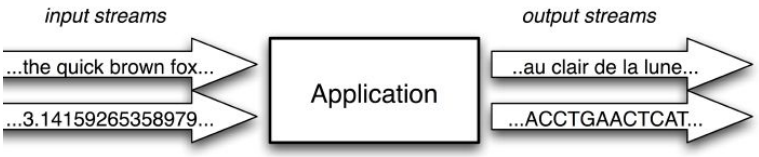
```
thick.addItemListener(e -> {
   if (e.getStateChange() == ItemEvent.SELECTED) {
      stroke = thickStroke;
   } else {
      stroke = thinStroke;
});
```

Method Argument(s)

```
SwingUtilities.invokeLater(() -> new PaintF());
```

I/O Streams

- The *stream* abstraction represents a communication channel with the outside world.
  - can be used to read or write a potentially unbounded number of data items (unlike a list)
  - data items are read from or written to a stream one at a time
- The Java I/O library uses subtyping to provide a unified view of disparate data sources and sinks.



Creating Streams (1)

- From a Collection via the stream() and parallelStream() methods;
- From an array via Arrays.stream(Object[]);
- The lines of a file can be obtained from BufferedReader.lines();
- Streams of file paths can be obtained from methods in Files;
- Streams of random numbers can be obtained from Random.ints();
- From static factory methods on the stream classes, such as Stream.of(Object[]), IntStream.range(int, int) or Stream.iterate(Object, UnaryOperator);
- Numerous other stream-bearing methods in the JDK, including BitSet.stream(), Pattern.splitAsStream(java.lang.CharSequence), and JarFile.stream().

Creating Streams (2)

- Can create your own Low-Level Stream
- Similar to having a custom class like `WordScanner` that implements `Iterator`
- `Spliterator` – parallel analogue to `Iterator`
  - (Possibly infinite) Collection of elements
  - Support for:
    - Sequentially advancing elements (similar to `next()`)
    - Bulk Traversal (performs the given action for each remaining element, ***sequentially*** in the current thread)
    - Splitting off some portion of the input into another spliterator, which can be processed in ***parallel*** (much easier than doing threads manually!)

Streams Operations

- Intermediate (Stream-producing) operations
  - Similar to transform in Ocaml
  - Return a new stream
  - Always lazy
  - Traversal of the source does not being until the terminal operation of the pipeline is executed
  - E.g., `filter`, `map`, `sorted`
- Terminal (value- or side-effect-producing) operations
  - Similar to fold in Ocaml
  - Produce a result or side-effect
  - E.g., `forEach`, `reduce`, `findFirst`, `allMatch`, `max`, `min`
- Combined to create Stream pipelines

Lambdas, Streams, Pipelines

- Beauty and Joy of functional programming, now in Java!

```
roster.stream()
      .filter(p ->
               p.getGender() == Person.Sex.MALE
               && p.getAge() >= 18
               && p.getAge() <= 25)
      .map(p -> p.getEmailAddress())
      .forEach(email -> System.out.println(email));
```

```
int sum = widgets.stream()
                 .filter(b -> b.getColor() == RED)
                 .mapToInt(b -> b.getWeight())
                 .sum();
```

- **L40 - Semester Recap - (467)**

Types, Generics, and Subtyping

Concept: *Static type systems* prevent errors. Every expressio has a static type, and OCaml/Java use the types to rule out buggy programs. *Generics* and *subtyping* make types more flexible and allow for better code reuse.

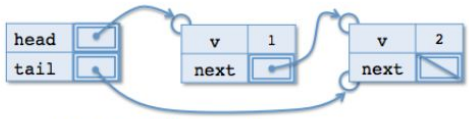
```
let rec contains (x:'a) (l:'a list) : bool =
  begin match l with
    | [] -> false
    | h::tl -> x = a || (contains x tl)
  end
```

Why?

- Easier to fix problems indicated by a type error than to write a test case and then figure out why the test case fails
- Promotes refactoring: type checking ensures that basic invariants about the program are maintained

Mutable data

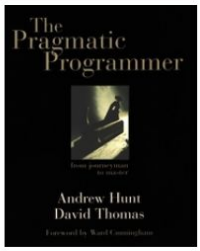
- Concept: Some data structures are *ephemeral*: computations mutate them over time
- Examples: queues, deques (HW4), GUI state (HW5, 9), arrays (HW 6), dynamic arrays, dictionaries (HW8)
- Why?
  - Common in OO programming, which simulates the transformations that objects undergo when interacting with their environment
  - Heavily used for event-based programming, where different parts of the application communicate via shared state
  - Default style for Java libraries (collections, etc.)

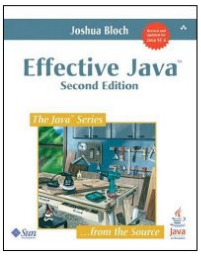


A queue with two elements

The Craft of Programming

- *The Pragmatic Programmer:*
  From Journeyman to Master
  by Andrew Hunt and David Thomas
  - Not about a particular programming language, it covers style, effective use of tools, and good practices for developing programs.

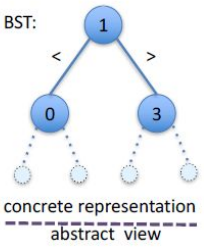




- *Effective Java*
  by Joshua Bloch
  - Technical advice and wisdom about using Java for building software. The views we have espoused in this course share much of the same design philosophy.

Abstract types and encapsulation

- Concept: *Type abstraction* hides the actual implementation of a data structure, describes a data structure by its interface (what it does vs. how it is represented), supports reasoning with invariants
- Examples: Set/Map interface (HW3), queues in nd access

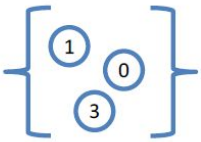


Invariants are a crucial tool for reasoning about data structures:

1. *Establish* the invariants when you create the structure.
2. *Preserve* the invariants when you modify the structure.

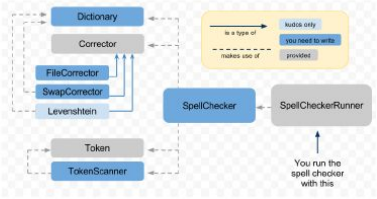
entation without

about the



Sequences, Sets, Maps

- Concept: Specific **abstract data types** of sequences, sets, and finite maps
- Examples: HW3, Java Collections, HW 7, 8
- Why?
  - These abstract data types come up again and again
  - Need aggregate data structures (collections) no matter what language you are programming in
  - Need to be able to choose the data structure with the right semantics



General Q's:

- Difference between static and non-static methods

https://github.com/younglee327/cis120notes/blob/master/Notes.md
https://github.com/younglee327/cis120notes/blob/master/TrueFalse.md

Code:

M3:

- Histogram.java
    - "".java
    - WordScanner.java
- DrawingCanvas.java
- DrawingCanvasMain.java
- OnOff (lightbulb demo)
- Layout.demo
- Multithreaded.java
- GCtest.java
- PaintA....E.java
- Streams.java