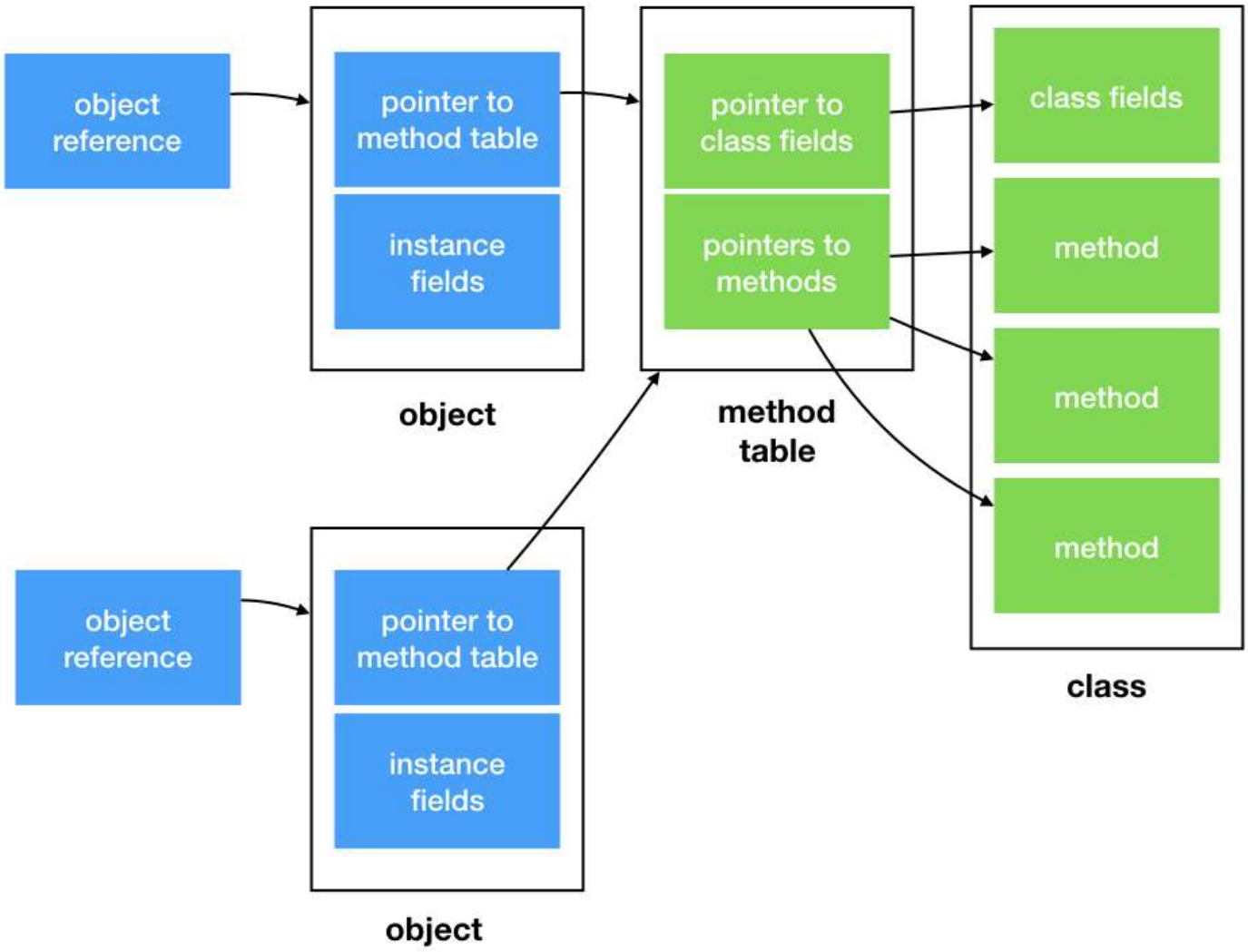
Lecture 1: Abstraction and Encapsulation

Memory Model for Objects

Variables and functions are stored in the memory of the computers as bits, usually in two separate regions. Since an object encapsulates both variables and functions, where are they stored?

Different implementations might stored the objects differently, but here is one way that we will follow for this class:



In the figure above, there are two objects of the same class. An objects is referred to through its references, which is a pointer to memory location where the instance fields for the object is stored, along with a pointer to a *method table*. A method table stores a table of pointers to the methods, along with a table to the class fields.

Lecture 2: Inheritance & Polymorphism

Interface as Types

In Java, an interface is a type. What this means is that:

- We can declare a variable with an interface type, such as:

```
    GeometricShape circle;
```

or

```
    Printable circle;
```

We cannot, however, instantiate an object from an interface since an interface is a "template", an "abstraction", and does not have an implementation. For instance:

```
    // this is not OK
    Printable p = new Printable();
    // this is OK
    Printable circle = new Circle(new Point(0, 0), 10);
```

- Similarly, we can pass arguments of an interface type into a method, and the return type of a method can be an interface.
- An object can be an instance of multiple types. Recall that Java is a statically typed language. We associate a type with a variable. We can assign a variable to an object if the object is an instance of the type of the variable. Line 4 above, for instance, creates a new circle, which is an instance of three types: `Circle`, `GeometricShape`, and `Printable`. It is ok to assign this new circle to a variable of type `Printable`.

We can now do something cool like this:

```
    Printable[] objectsToPrint;
      :
      // initialize array objectsToPrint
    :
    for (Printable obj: objectsToPrint) {
        obj.print();
    }
```

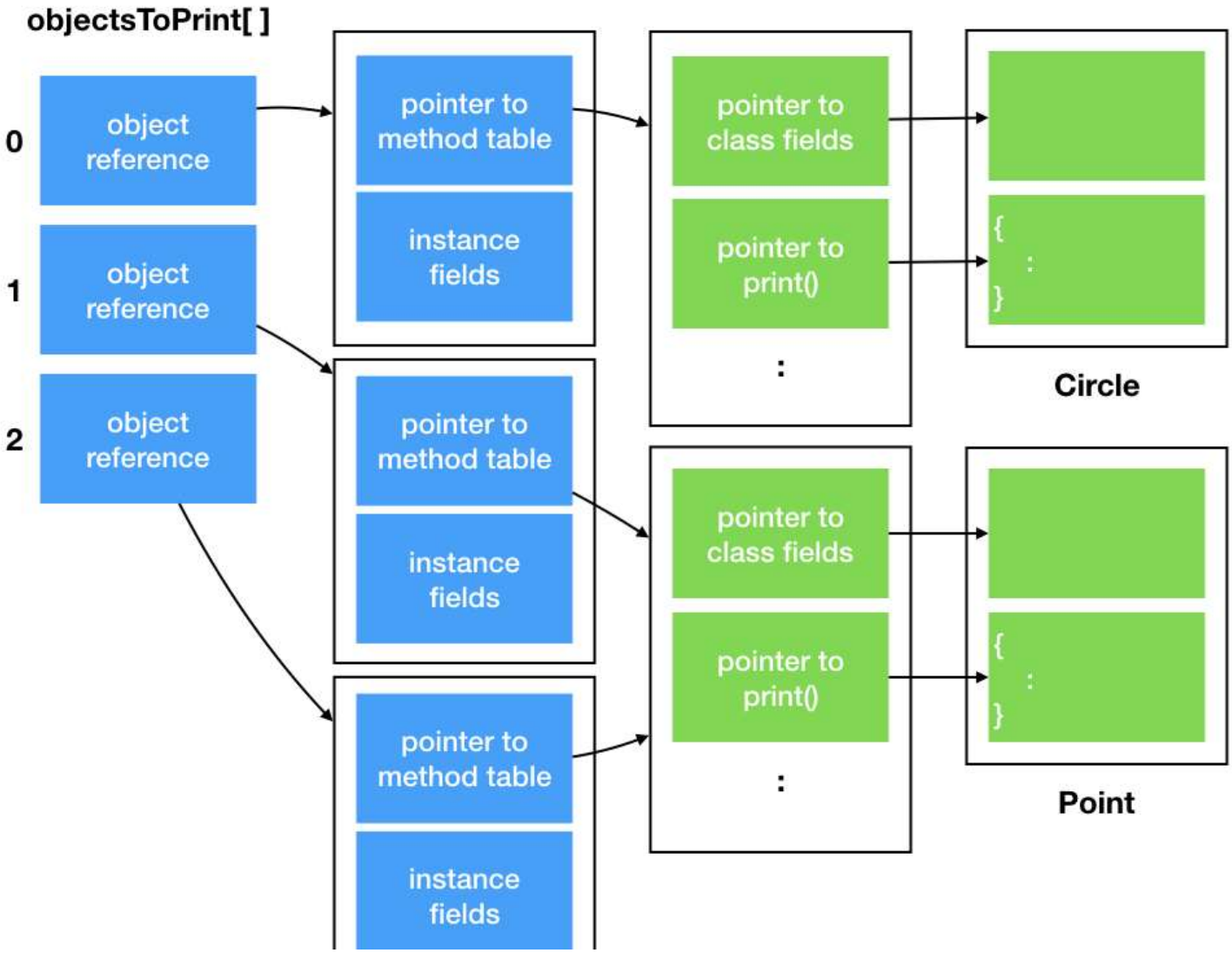
Let's look at this code in more details. Line 1 declares an array of objects of type `Printable`. We skip over the code to initialize the content of the array for now, and jump to Line 5-7, which is a `for` loop. Line 5 declares a loop variable `obj` of type `Printable` and loops through all objects in the array `objectsToPrint`, and Line 6 invoke the method `print` of `obj`.

The magic happens in Line 6:

- First, since we now that any object in the array has the type `Printable`, this means that they must implement the `Printable` interface and support the method `print()`.
- Second, we do not know, and we do not *need* to know which class an object is an instance of.
- Third, we can actually have objects of completely unrelated classes in the same array. We can have objects of type `Circle`, and objects of type `Point`. We can have objects of type `Factory`, or objects of type `Student`, or objects of type `Cushion`. As long as the objects implement the `Printable` interface, we can put them into the same array.
- Forth, at *run time*, Java looks at `obj`, and determines its class, and invoke the right implementation of `print()` corresponding to the `obj`. That is, if `obj` is an instance of a class `Circle`, then it will call `print()` method of `Circle`; if `obj` is an instance of a class `Point`, then it will call `print()` method of `Point`, and so on.

To further appreciate the magic that happens in Line 6, especially the last point above, consider how function call is done in C. In C, you cannot have two functions of the same name within the same scope, so if you call a function `print()`, you know exactly which set of instructions will be called. So, the name `print` is bound to the corresponding set of instructions at compilation time. This is called *static binding* or *early binding*. To have `print()` for different types, we need to name them differently to avoid naming conflicts: e.g., `print_point()`, `print_circle()`.

In OO languages, you can have methods named `print()` implemented differently in different classes. When we compile the code above, the compiler will have no way to know which implementation will be called. The bindings of `print()` to the actual set of instructions will only be done at run time, after `obj` is instantiated from a class. This is known as *dynamic binding*, or *late binding*, or *dynamic dispatch*.



If you understand how an object is represented internally, this is not so magical after all. Referring to the figure above, the array `objectsToPrint[]` contains an array of references to objects, the first one is a `Circle` object, and the next two are `Point` objects. When `obj.print()` is invoked, Java refers to the method table, which points to either the method table for `Circle` or for `Point`, based on the class the object is an instance of.

This behavior, which is common to OO programming languages, is known as *polymorphism*.

Java `Object` class

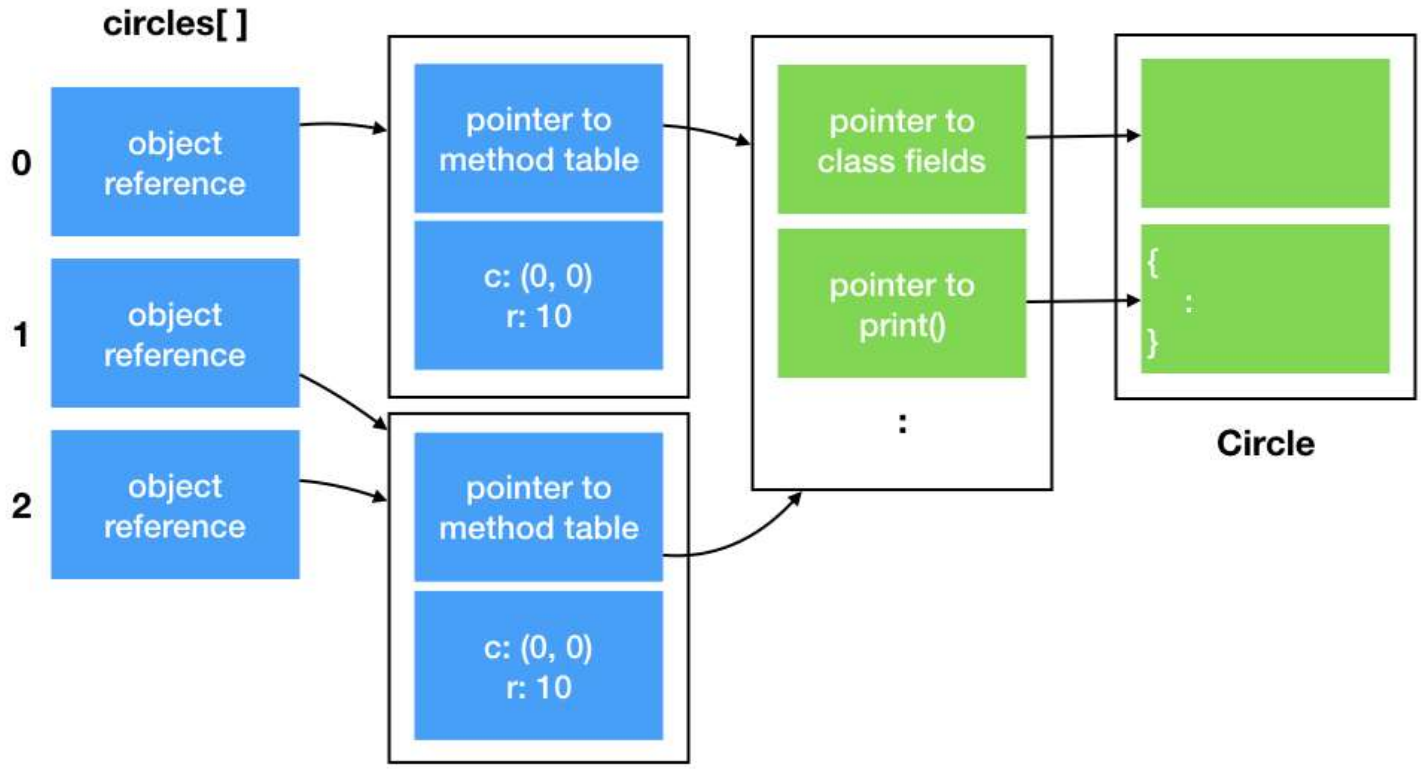
In Java, every class inherits from the class `Object` [https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html] implicitly. The `Object` class defines many useful methods that are common to all objects. The two useful ones are :

- `equals(Object obj)`, which checks if two objects are equal to each other, and
- `toString()`, which returns a string representation of the object, and is a better way to print an object than the `print()` method and `Printable` interface we write.

The `equals()` method as implemented in `Object`, only compares if two object references refer to the same object In the Figure below, we show an array `circles` with three `Circle` objects. All three circles are centered at (0, 0) with radius 10. They are created as follows:

```
Circle[] circles = new Circle[3];
circles[0] = new Circle(new Point(0, 0), 10);
circles[1] = new Circle(new Point(0, 0), 10);
circles[2] = circles[1];
```

When you check `circles[0].equals(circles[1])`, however, it returns `false`, because even though `circles[0]` and `circles[1]` are semantically the same, they refer to the two different objects. Calling `circles[1].equals(circles[2])` returns `true`, as they are referring to the same object.



The `equals()` method is universal (all classes inherits this method) and is used by other classes for equality tests. So, in most cases, we can implement a method called `equals()` with the same signature with the semantic that we want [1] [#fn:8].

The method in the subclass will override the method in the superclass. For example,

```
class Circle implements Shape, Printable {
 :
    @Override
    public boolean equals(Object obj) {
        if (obj instanceof Circle) {
            Circle circle = (Circle) obj;
            return (circle.center.equals(center) && circle.radius == radius);
        } else
```

```
9                   return false;
10              }
11          }
```

- Line 4 declares the method `equals`, and note that it has to have exactly the same signature as the `equals()` method we are overriding. Even though we meant to compare two `Circle` objects, we cannot declare it as `public boolean equals(Circle circle)`, since the signature is different and the compiler would complain.
- Since `obj` is of an `Object` type, we can actually pass in any object to compare with a `Circle`. Line 5 checks if the comparison makes sense, by checking if `obj` is instantiated from a `Circle` class, using the `instanceof` keyword. If `obj` is not even a `Circle` object, then we simply return `false`.

For the code above to work, we have to override the `equals` method of `Point` as well.

One final note: polymorphism works here as well. If we have an object reference `obj` of type `Object` that refers to an instance of a `Circle`, calling `obj.equals()` will invoke the `equals()` method of `Circle`, not `Object`, just like the case of interfaces.

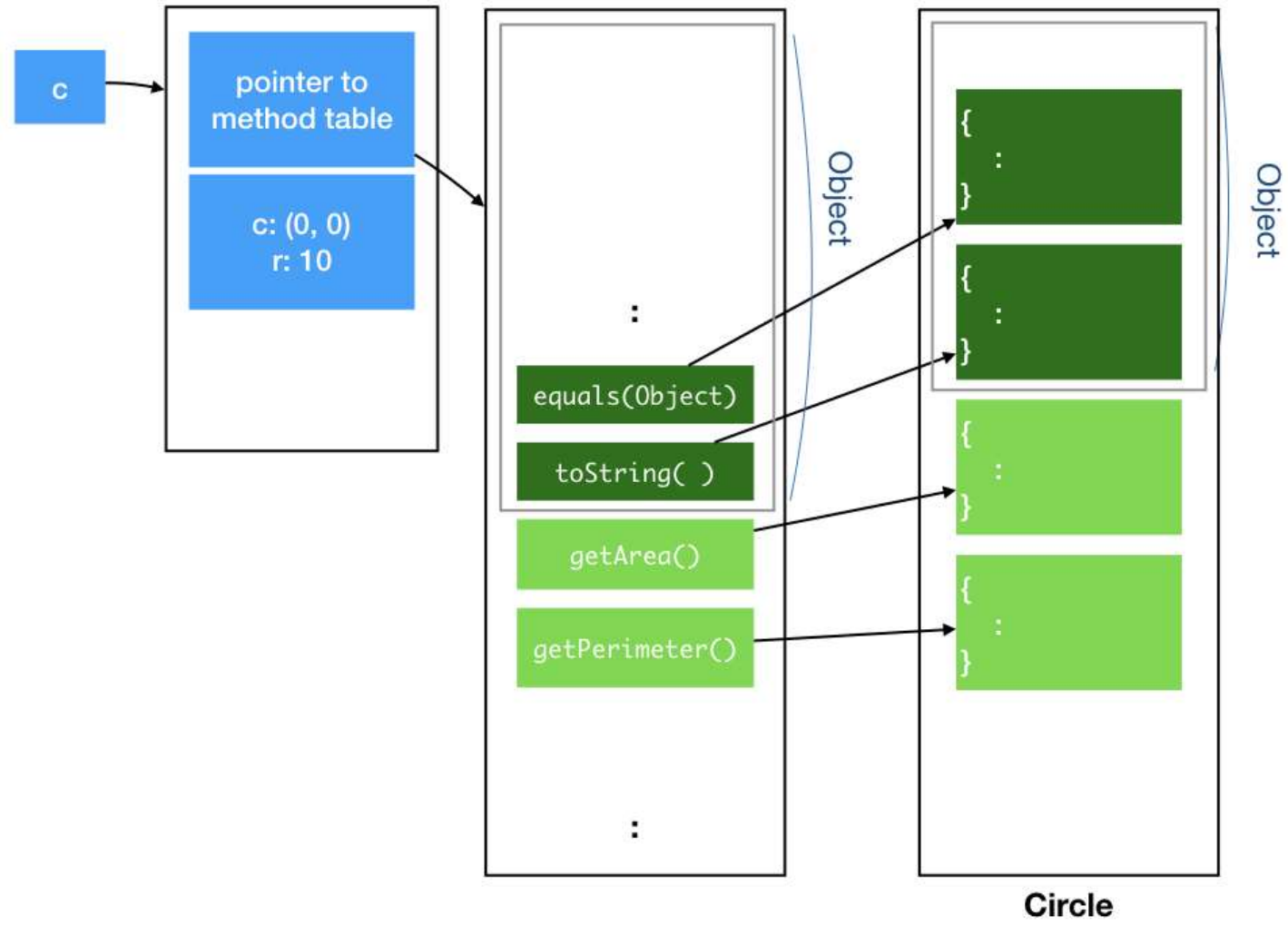
---

1. If you override `equals()` you should generally override `hashCode()` as well, but let's leave that for another lesson on another day.

Lecture 3: Inheritance, Continued

Method Overriding

The figure below illustrates this. I use a slightly darker green to represent the methods implemented in the `Object` class, and light green to represent methods implemented in the `Circle` class.

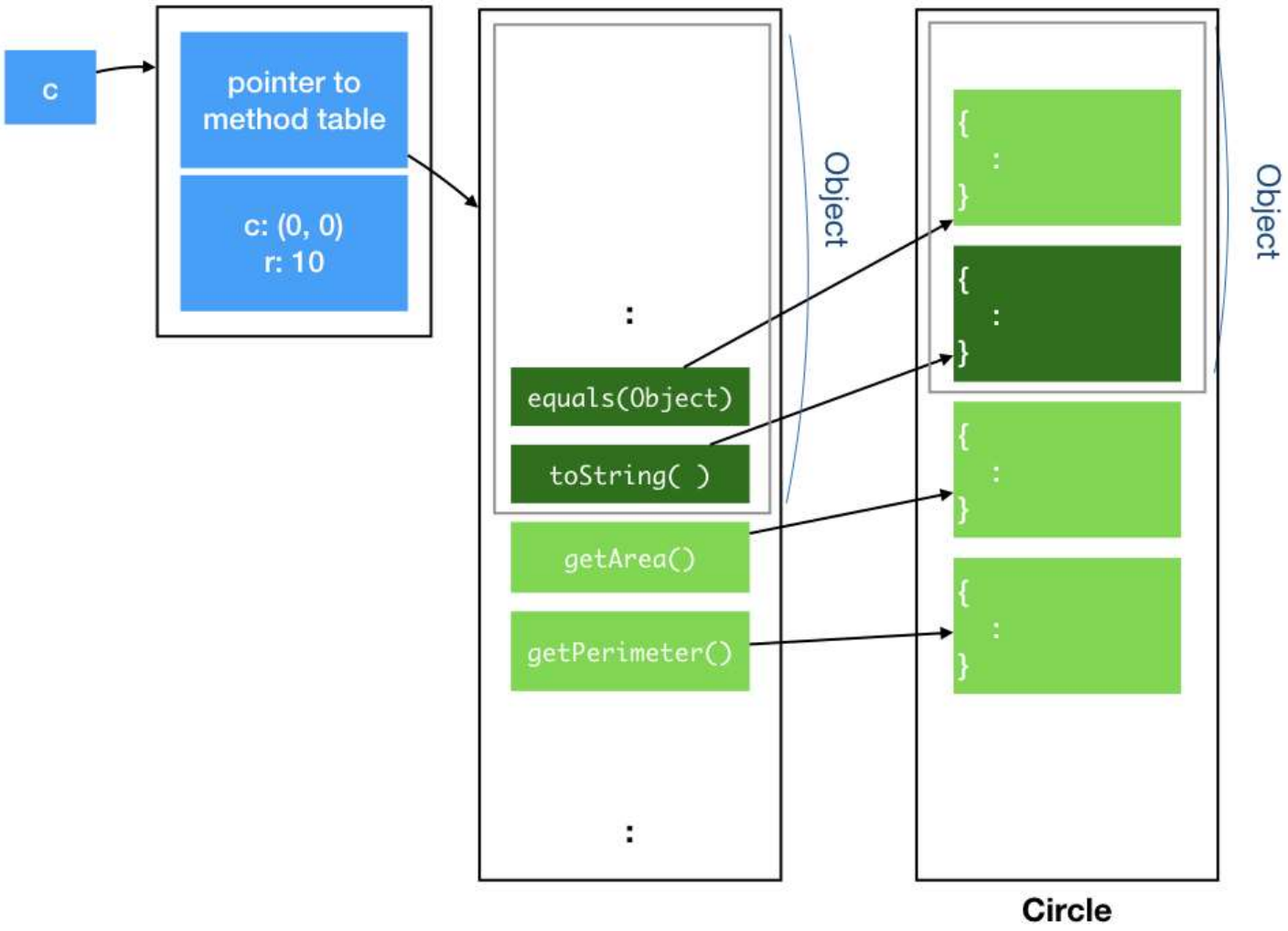


Now, consider what would happen if we override the method `equals()` from the `Object` class. This is what we did in the last lecture.

```
1    class Circle implements Shape, Printable {
2     :
3        @Override
4        public boolean equals(Object obj) {
5            if (obj instanceof Circle) {
6                Circle circle = (Circle) obj;
```

```
            return (circle.center.equals(center) && circle.radius == radius);
        } else
            return false;
    }
}
```

The method table will update the entry for `equals()` to point to the implementation provided by the `Circle` class, instead of the `Object` class.



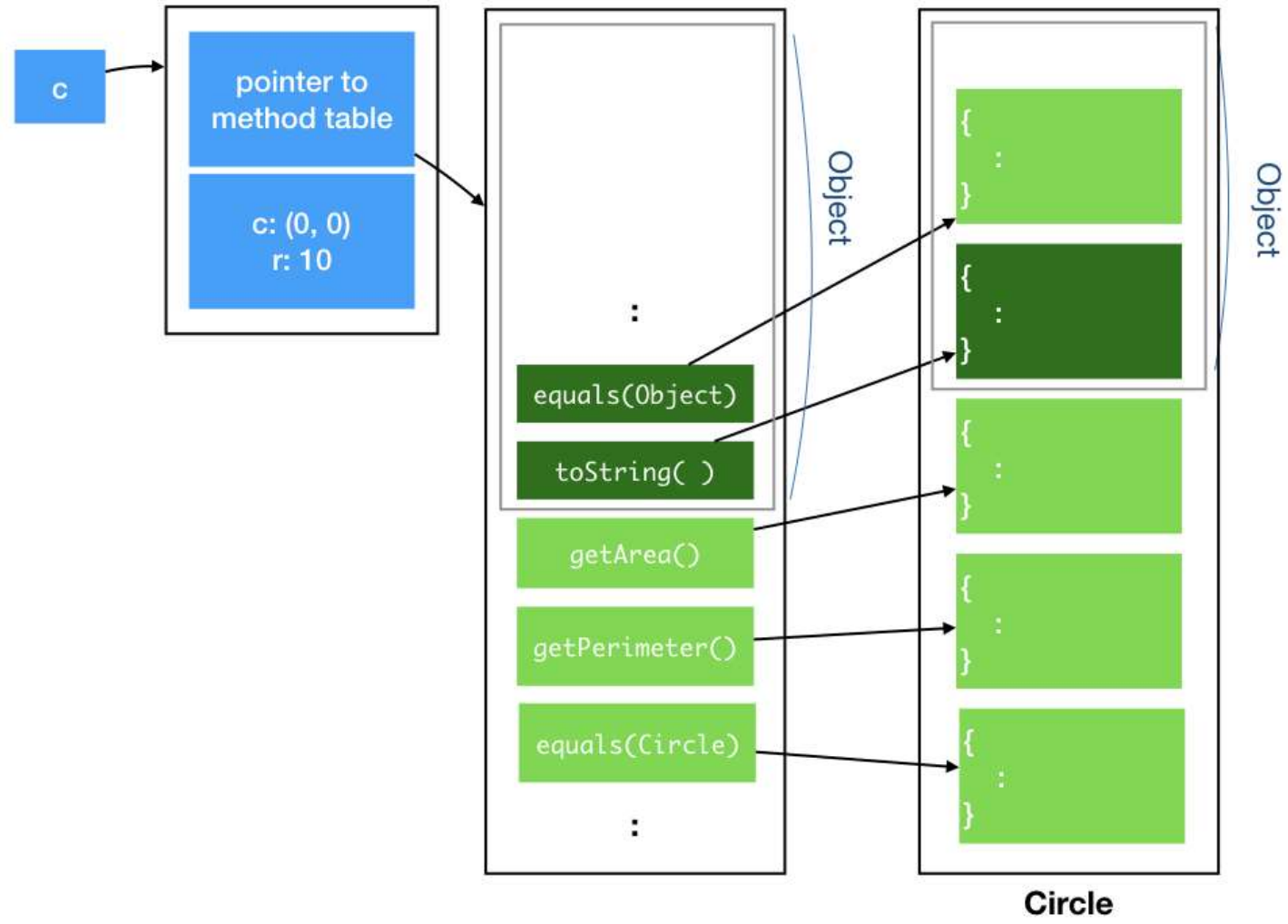
Now, consider what would happen if we *overload* the method `equals()` with one that takes in a `Circle` object. I also throw in a couple of `System.out.print()` to help us figure out what is going on.

```
class Circle implements Shape, Printable {
 :
    @Override
    public boolean equals(Object obj) {
        System.out.print("equals(Object) called\n");
        if (obj instanceof Circle) {
            Circle circle = (Circle) obj;
            return ((circle.center.equals(center) && circle.radius == radius);
        } else
            return false;
    }

    public boolean equals(Circle circle) {
```

```
            System.out.print("equals(Circle) called\n");
            return ((circle.center.equals(center) && circle.radius == radius);
        }
    }
```

Since this new `equals()` method does not override the method in `Object`, it gets its own slot in the method table of `Circle`, instead of reusing one from the `Object`.



Now, consider which version of `equals` are called by the following:

```
Circle c1 = new Circle(new Point(0,0), 10);
Circle c2 = new Circle(new Point(0,0), 10);
Object o1 = c1;
Object o2 = c2;

o1.equals(o2);          //Circle.equals(Object) [Overriden in Circle]
o1.equals((Circle)o2);  //Circle.equals(Object)
o1.equals(c2);          //Circle.equals(Object) [Object has no equals(Circle)]
c1.equals(o2);          //Circle.equals(Object)
c1.equals((Circle)o2);  //Circle.equals(Circle)
c1.equals(c2);          //Circle.equals(Circle)
```

Another question that came up is why we need to override `equals` in `Object`, rather than just using the `Circle`-specific `equals(Circle)`. As shown above, only when an object declared as `Circle` calls `equals`

on another `Circle` object, the `Circle`-specific `equals(Circle)` is invoked.

To write code that is general and reusable, we should exploit OO polymorphism, that means different subclasses of `Object` implement their own customized version of `equals`, and the right version of `equals` will be called.

We now turn our attention to another method in `Object` that we could override, the `toString()` method. `toString()` is called whenever the `String` representation of an object is needed.

```
class Point {
    :
    public String toString() {
        return "(" + x + "," + y ")";
    }
}
```

Now, if we run:

```
Point p = new Point(0,0);
System.out.println(p);
```

It should print `(0,0)` instead of `Point@1235de`.

Modeling HAS-A Relationship

Composition allows us to build more complex classes from simpler ones, and is usually favored over inheritance.

The `PaintedShape` class from Lecture 2, for instance, could be modeled as a composition of a `Style` object and `GeometricShape` object.

```
class Style {
    Color fillColor;
    Color borderColor;
      :
}

class PaintedShape {
    Style style;
    GeometricShape shape;
      :
    public double getArea() {
        return shape.getArea();
    }
    :
    public void fillWith(Color c) {
        style.fillWith(c);
    }
      :
}
```

The design above is also known as the *forwarding* -- calls to methods on `PaintedShape` gets forwarded to either `Style` or `GeometricShape` objects.

Modeling IS-A Relationship

A better situation to use inheritance is to model a IS-A relationship: when the subclass behaves just like parent class, but has some additional behaviors. For instance, it is natural to model a `PaintedCircle` as a subclass of `Circle` -- since a PaintedCircle has all the behavior of `Circle`, but has *additional* behaviors related to being painted.

```
class PaintedCircle extends Circle {
    Style style;
      :
}
```

It is a developer's responsibility that any inheritance with method overriding does not alter the behavior of existing code. This brings us to the Liskov Substitution Principle, which says that: "Let be a property provable about objects of type . Then should be true for objects of type where is a subtype of ." This means that if is a subclass of , then an object of type can be replaced by an object of type without changing the desirable property of the program.

This means that everywhere we can expect rectangles to be used, we can replace a rectangle with a square. This was no longer true with the introduction of `resizeTo` method.

Preventing Inheritance and Method Overriding

Both the two java classes you have seen, `java.lang.Math` and `java.lang.String`, cannot be inherited from. In Java, we use the keyword `final` when declaring a class to tell Java that we ban this class from being inherited.

```
final class Circle {
    :
}
```

Alternatively, we can allow inheritance, but still prevent a specific method from being overridden, by declaring a method as `final`. Usually, we do this on methods that are critical for the correctness of the class.

```
class Circle {
   :
    final public boolean contains(Point p) {
        :
    }
}
```

The keyword `final` has another use. When declaring a variable as `final`, just like `PI` in `Math`, it prevents the variable from being modified. In other words, the variable becomes constant.

```
    public static final double PI   = 3.141592653589793;
```

Abstract Class and Interface with Default Methods

- An *abstract class*, which is just like a class, but it is declared as `abstract`, and some of its methods are declared as `abstract`, without implementation. An abstract class cannot be instantiated, and any subclass who wish to be concrete needs to implement these abstract methods.

```
abstract class PaintedShape {
    Color fillColor;
      :
    void fillWith(Color c) {
        fillColor = c;
    }
    :
    abstract double getArea();
    abstract double getPerimeter();
      :
```

- An interface with default implementation. Introduced only in Java 8, with the goal of allowing interface to evolve, an interface can now contain implementation of the methods. Such interface still cannot be instantiated into objects, but classes that implement such interface need not provide implementation for a method where a default implementation exists. For instance, we can have:

```
interface GeometricShape {
    public double getArea();
    public double getPerimeter();
    public boolean contains(Point p);
    default public boolean cover(Point p) {
        return contains(p);
    }
    :
```

  where `cover` is a new method with default implementation, denoted with keyword `default`.

At this point in CS2030, let's not worry about when to use abstract class or default methods in interfaces, but just be aware that they exists and understand what they mean when you come across them. After you gain some experience writing OO programs, we will revisit these concepts so that you can better appreciate their differences and usage.

After Note

- When we override `equals()` of `Object` in `Circle`, I said "*The method table will update the entry for `equals()` to point to the implementation provided by the `Circle` class, instead of the `Object` class.*" and in the figure, I showed that the code for `Circle`'s customized `equals` replacing the `equals` for `Object`. What I should have added, is that, the original implementation of `equals` from `Object` is not completely gone. Methods from the immediate parent that have been overriden can still be called, with `super` keyword. Here is a useful example from `Point`'s `toString()`:

```
@Override
public String toString() {
  return super.toString() + " (" + x + "," + y + ")";
}
```

which prefix the string representation of Point with the class and reference address.

- Additional notes

```
            | Class | Package | Subclass | Subclass | World
            |       |         |(same pkg)|(diff pkg)|
------------+-------+---------+----------+----------+--------
public      |   +   |    +    |    +     |     +    |   +
------------+-------+---------+----------+----------+--------
protected   |   +   |    +    |    +     |     +    |
------------+-------+---------+----------+----------+--------
no modifier |   +   |    +    |    +     |          |
------------+-------+---------+----------+----------+--------
private     |   +   |         |          |          |

+ : accessible
blank : not accessible
```

Lecture 4: Memory, Exception and Generics

Stack and Heap

Objects are created on the heap and Java stores references to them. Primitives and object references are created on the stack.

Call Stack

Java uses *call by value* for primitive types, and *call by reference* for objects.

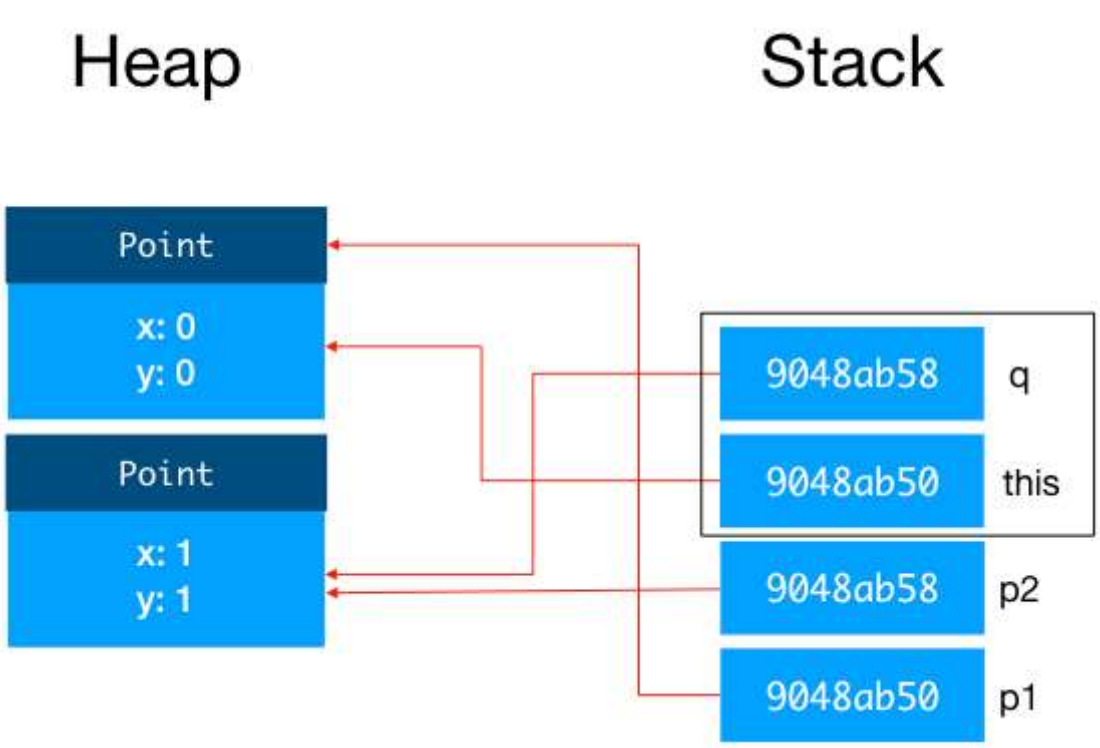
Now, let's look at what happens when we invoke a method. Take the `distanceTo` method in `Point` as an example:

```
class Point {
  private double x;
  private double y;
  public double distanceTo(Point q) {
    return Math.sqrt((q.x - this.x)*(q.x - this.x)+(q.y - this.y)*(q.y - this.y));
  }
}
```

and the invocation:

```
Point p1 = new Point(0,0);
Point p2 = new Point(1,1);
p1.distanceTo(p2);
```

When `distanceTo` is called, Java (to be more precise, the Java Virtual Machine, or JVM) creates a *stack frame* for this instance method call. This stack frame is a region of memory that tentatively contains (i) the `this` reference, (ii) the method arguments, and (iii) local variables within the method, among other things[3][4]. When a class method is called, the stack frame does not contain the `this` reference.

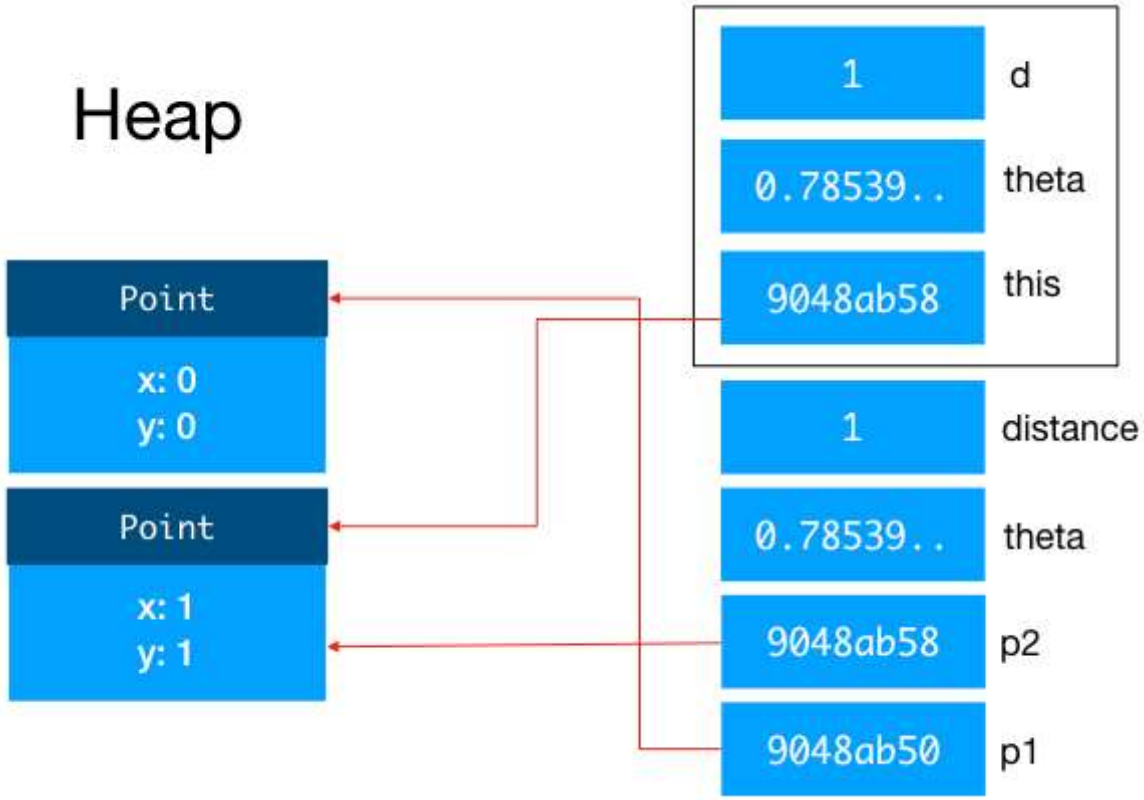


You can see that the *reference* of the objects `p1` and `p2` are copied onto the stack frame. `p1` and `this` point to the same object, and `p2` and `q` point to the same object. Within the method, any modification done to `this` would change the object referenced to by `p1`, and any change made to `q` would change the object referenced to by `p2` as well. After the method returns, the stack frame for that method is destroyed.

Let's call the `move` function from your Lab 1 [lab1.md], with arguments `(double theta, double d)`.

```
1  double theta = Math.PI/4.0;
2  double distance = 1;
3  p2.move(theta, distance);
```

Again, we create a stack frame, copy the reference to object `p2` into `this`, copy `theta` from the calling function to `theta` the argument within the method, copy `distance` from the calling function to `d` the argument within the method. Recall that, in this function, you actually change the `x` and `y` of `this` to move `p2`.



What is important here is that, as `theta` and `distance` are primitive types instead of references, we copy the values onto the stack. If we change `theta` or `d` within `move`, the `theta` and `distance` of the calling function will not change. If you want to pass in a variable of primitive type into a method and have its value changed, you will have to use a *wrapper class*.

If we made multiple nested method calls, as we usually do, the stack frames get stacked on top of each other.

One final note: the memory allocated on the stack are deallocated when a method returns. The memory allocated on the heap, however, stays there as long as there is a reference to it (either from another object or from a variable in the stack). In Java, you do not have to free the memory allocated to objects. The JVM runs a *garbage collector* that checks for unreferenced objects on the heap and cleans up the memory automatically.

Exceptions

Java supports `try` / `catch` / `finally` control statements, which is a way to group statements that check/handle errors together making code easier to read. The Java equivalent to the above is:

```
try {
  reader = new FileReader(filename);
  scanner = new Scanner(reader);
  numOfPoints = scanner.nextInt();
}
catch (FileNotFoundException e) {
    System.err.println("Unable to open " + filename + " " + e);
}
catch (InputMismatchException e) {
    System.err.println("Unable to scan for an integer");
}
catch (NoSuchElementException e) {
```

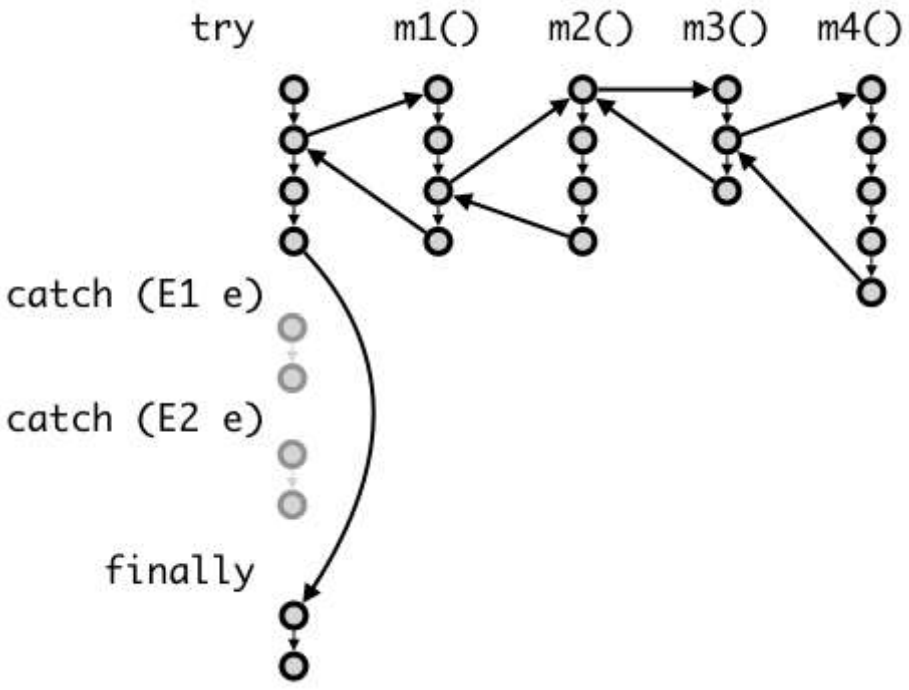
```
    System.err.println("No input found");
}
finally {
  if (scanner != null)
    scanner.close();
}
```

"Error vs. Exception in Java"
- We have been using the term error and exception loosely. Java has different classes for `Error` and `Exception`. `Error` is for situations where the program should terminate as generally there is no way to recover. For instance, when the heap is full (`OutOfMemoryError`) or the stack is full (`StackOverflowError`). Exceptions are for situations where it is still possible to reasonably recover from the error.

"Combining Multiple Catches" - In cases where the code to handle the exceptions is the same, you can

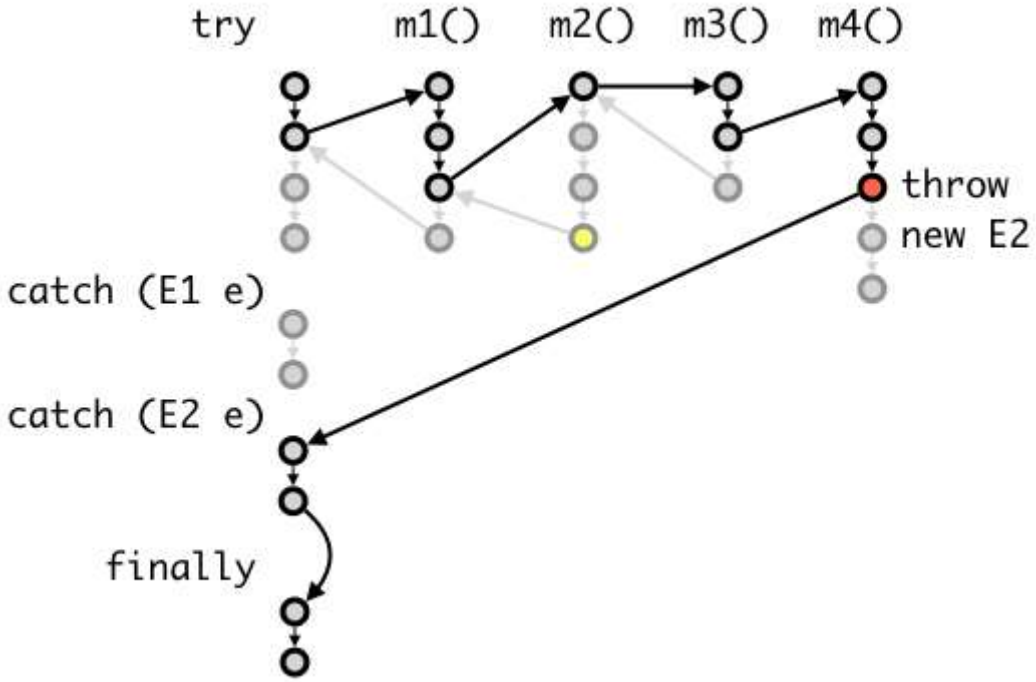
```
catch (FileNotFoundException | InputMismatchException | NoSuchElementException e) {
    System.err.println(e);
}
```

Here is a more detailed description of the control flow of exceptions. Consider we have a `try` - `catch` - `finally` block that catches two exceptions `E1` and `E2`. Inside the `try` block, we call a method `m1()`; `m1()` calls `m2()`; `m2()` calls `m3()`, and `m3()` calls `m4()`. In a normal (no exception) situation, the control flow looks like this:



The statements in `try` block is executed, followed by the statements in `finally` block.

Now, let's suppose something went wrong deep inside the nested call, in `m4()`. One of the statement executes `throw new E2();`, which causes the execution in `m4()` to stop. JVM now looks for the block of code that catches `E2`, going down the call stack, until it can find a place where the exception is handled. In this example, we suppose that none of `m1()` - `m4()` handles (i.e., `catch`) the exception. Thus, JVM then jumps to the code that handles `E2`. Finally, JVM executes the `finally` block.



Checked Exceptions

There are two types of exceptions in Java: *checked* and *unchecked* exceptions:

- A checked exception is something that the programmer should anticipate and handle. For instance, when you open a file, you should anticipate that in some cases, the file cannot be open.
- An unchecked exception is something that the programmer does not anticipate, and usually is a result of bugs. For example, when you try to call `p.distanceTo(q)` but `p` is `null`, resulting in a `NullPointerException` being thrown.

We need to catch all checked exceptions or let it propagate to the calling method. Otherwise, the program will not compile.

For unchecked exceptions, even though we could catch it, it makes more sense to eliminate the bugs. In Java, unchecked exceptions are subclasses of `RuntimeException`. All `Error`s are unchecked.

All methods that throw checked exception need to *specify* the checked exception(s). For example, if we want to put the code to open a file and read an `int` from it into a function, and want the calling function to deal with the exception, this is what we should do:

```
1  public static int readIntFromFile(String filename)
2    throws FileNotFoundException {
3    FileReader reader = new FileReader(filename);
4    Scanner scanner = new Scanner(reader);
5    int numOfPoints = scanner.nextInt();
6    scanner.close();
7    return numOfPoints;
8  }
```

Note Line 2 specify that this method might throw `FileNotFoundException`.

A checked exception must be either caught or thrown to calling function, except `main`, which has no calling function to throw to. If the main() does not catch an checked exception, the running program exits, and the exception is revealed to the user -- this is generally considered as bad programming.

The two other exceptions from the examples above `InputMismatchException` and `NoSuchElementException` are subclasses of `RuntimeException`, and therefore are unchecked.

Generating Exception

The Circle constructor in Lab 1 requires the distance between two input points to be . If the condition is violated, you are asked to return an invalid circle. Another way is to throw an unchecked exception `IllegalArgumentException` if one of the above two conditions is met.

```
public Circle(Point p, Point q, double r, boolean centerOnLeft) {
  if (p.distanceTo(q) > 2*r) {
    throw new IllegalArgumentException("Input points are too far apart");
  }
  if (p.equals(q)) {
    throw new IllegalArgumentException("Input points coincide");
  }
```

Note that difference between `throw` and `throws`: the former is to generate an exception, the latter to specify that the exception(s) thrown by a method.

Overriding Method that Throws Exceptions

When you override a method that throws a checked exception, the overriding method must throw only the same, or a more specific checked exception, than the overridden method. This rule enforces the Liskov Substitution Principle. The caller of the overridden method cannot expect any new checked exception than what has already been "promised" in the method specification.

Good Practices for Exception Handling

Catch Exceptions to Clean Up

While it is convenient to just let the calling method deals with exceptions ("Hey! Not my problem!"), it is not always responsible to do so. Consider the example earlier, where `m1()`, `m2()`, and `m3()` do not handle exception E2. Let's say that E2 is a checked exception, and it is possible to react to this and let the program continues properly. Also, suppose that `m2()` allocated some system resources (e.g., temporary files, network connections) at the beginning of the method, and deallocated the resources at the end of the method. Not handling the exception, means that, code that deallocates these resources does not get called when an exception occur! It is better for `m2()` to catch the exception, handle the resource deallocation in a `finally` block. If there is a need for the calling methods to be aware of the exception, `m2()` can always re-throw the exception:

```
public void m2() throws E2 {
```

```
    try {
      // setup resources
      m3();
    }
    catch (E2 e) {
      throw e;
    }
    finally {
      // clean up resources
    }
  }
```

Note that the `finally` block is always executed even when `return` or `throw` is called in a `catch` block.

Catch All Exception is Bad

DO NOT silently ignore thrown exceptions to shut the compiler up:

```
try {
  // your code
}
catch (Exception e) {}
```

```
try {
  // your code
}
catch (Error e) {}
```

Do not exit a program just because of exception. This would prevent the calling function from cleaning up their resources. Worse, do not exit a program silently.

```
try {
  // your code
}
catch (Exception e) {
  System.exit(0);
}
```

Do Not Break Abstraction Barrier

Sometimes, letting the calling method handles the exception causes the implementation details to be leak, and make it harder to change the implementation later.

For instance, suppose we design a class `ClassRoster` with a method `getStudents()`, which reads the list of students from a text file.

```
class ClassRoster {
    :
  public Students[] getStudents() throws FileNotFoundException {
    :
  }
}
```

Later, we change the implementation to reading the list from an SQL database,

```
class ClassRoster {
    :
  public Students[] getStudents() throws SQLException {
    :
  }
}
```

We should, as much as possible, handle the implementation specific exceptions within the abstraction barrier.

Generics

Third topic of today is on generics.

Java 5 introduces generics, which is a significant improvement to the type systems in Java. It allows a *generic class* of some type `T` to be written:

```
class Queue<T> {
  private T[] objects;
    :
  public Queue<T>(int size) {...}
  public boolean isFull() {...}
  public boolean isEmpty() {...}
  public void enqueue(T o) {...}
  public T dequeue() {...}
}
```

`T` is known as *type parameter*. The same code as before can be written as:

```
Queue<Circle> cq = new Queue<Circle>(10);
cq.enqueue(new Circle(new Point(0, 0), 10));
cq.enqueue(new Circle(new Point(1, 1), 5));
Circle c = cq.dequeue();
```

Here, we passed `Circle` as *type argument* to `T`, creating a *parameterized type* `Queue<Circle>`.

"Diamond Notation"

We can use the short form `<>` in the constructor as the compiler can infer the type:

```
Queue<Circle> cq = new Queue<>(10);
```

We can use parameterized type anywhere a type is used, including as type argument. If we want to have a queue of queue of circle, we can:

```
Queue<Queue<Circle>> cqq = new Queue<>(10);
```

Lecture 5: Numbers, Strings, Collections

Wrapper Classes

Java provides a set of wrapper class: one for each primitive type: `Boolean`, `Byte`, `Character`, `Integer`, `Double`, `Long`, `Float`, and `Short`.

```
1  Queue<Integer> iq = new Queue<Integer>(10);
2  cq.enqueue(new Integer(4));
3  cq.enqueue(new Integer(8));
4  cq.enqueue(new Integer(15));
```

Java 5 introduces something called *autoboxing* and *unboxing*, which creates the wrapper objects automatically (autoboxing) and retrieves its value (unboxing) automatically. With autoboxing and unboxing, we can just write:

```
1  Queue<Integer> iq = new Queue<Integer>(10);
2  cq.enqueue(4);
3  cq.enqueue(8);
4  cq.enqueue(15);
```

Note that `enqueue` expects an `Integer` object, but we pass in an `int`. This would cause the `int` variable to automatically be boxed (i.e., be wrapped in Integer object) and put onto the call stack of `enqueue`.

Type Erasure

The reason why Java compiler does not allow generic class with primitive types, is that internally, the compiler uses *type erasure* to implement generic class. Type erasure just means that during compile time, the compiler replaces the type parameter with the most general type. In the example given in Lecture 4 [../lec4/index.html], `E` in `Queue<E>` is replaced with `Object`, The compiler then inserts necessary cast to convert the `Object` to the type argument (e.g., `Circle`), exactly like how it is done in the `ObjectQueue` example, and additional checks to ensure that only objects of given type is used as `E` (e.g., cannot add `Point` to `Queue<Circle>`). Since primitive types are not subclass `Object`, replacing `E` with primitive types would not work with type erasure.

Note that, due to type erasure at compile time, Java has no information about `E` at runtime.

In short, wrapper class allows us to use primitive types to parameterize a generic class, and we do not have to write code to box and unbox the primitive types.

Performance Penalty

If the wrapper class is so great, why not use it all the time and forget about primitive types?

The answer: performance. Because using an object comes with the cost of allocating memory for the object and collecting of garbage afterwards, it is less efficient than primitive types.
The primitive type is 2 times faster! Due to autoboxing and unboxing, the cost of creating objects become hidden and often forgotten.

All primitive wrapper class objects are immutable. What this means is that once you create an object, it cannot be changed. Thus, everytime `sum` in the example above is updated, a new object gets created!

String and StringBuilder

Another place with hidden cost for object creation and allocation is when dealing with `String`.

A `String` object is also *immutable*. When we do:

```
String words = "";
words += "Hello ";
words += "World!";
```

A new `String` object is created everytime we concatenate it with another `String`.

Java provides a mutable version of `String`, called `[StringBuilder](https://docs.oracle.com/javase/8/docs/api/java/lang/StringBuilder.html)`. To build up a string, we could do:

```
StringBuilder wordsBuilder = new StringBuilder();
wordsBuilder.append("Hello ");
wordsBuilder.append("World!");
String words = wordsBuilder.toString();
```

Code that uses `StringBuilder` is not as intuitive and readable than just using the `+` operator. My preference is to use `String` and `+` for occasional concatenation, and `StringBuilder` for frequent concatenation that could become performance bottleneck.

Equality for Strings and Wrapper Classes

Remember that `==` compares only references: whether the two references are pointing the the same object or not. The `equals` method has been overridden to compare if the values are the same or not. So, the right way to compare two strings or two numbers are:

```
if (s1.equals(s2)) { ... }
if (i1.equals(i2)) { ... }
```

If you try:

```
Integer i1 = 1;
```

```
Integer i2 = 1;
if (i1 == i2) { ... }
```

It might return `true`! This behaviour is caused by some autoboxing optimization in the Integer class so that it does not create too many objects for frequently requested values. It is called `Integer caching`. If another `Integer` object with the same value has been autoboxed before, JVM just returns that object instead of returning a new one. Do not rely on Integer caching for proper comparisons of `==`. Use `equals()`, always.

Similarly, if you try:

```
String s1 = "hello";
String s2 = "hello";
if (s1 == s2) { ... }
```

Java always returns `true`. This is because, the Java `String` class internally maintain a pool of *interned string* objects for all string literals and expression, as an optimization.

Java Collections

Now, we turn our attention to the Java Collection Framework. Java provides a rich set of classes for managing and manipulating data. They efficiently implement many useful data structures (hash tables, red black trees, etc.) and algorithms (sorting, searching, etc.) so that we no longer have to. As computer scientists, it is still very important for us to know how these data structures and algorithms can be implemented, how to prove some behaviors (such as running time) and their correctness, how certain trade offs are made, etc. They are so important that we have two modules dedicated to them: CS2040 and CS3230 in the core CS curriculum.

For CS2030, however, we focus on how to use them.

Collection

One of the basic interface in Java Collection Framework is `Collection<E>`, it looks like:

```
public interface Collection<E> extends Iterable<E> {
  boolean add(E e);
  boolean contains(Object o);
  boolean remove(Object o);
  void clear();
  boolean isEmpty();
  int size();

  boolean equals(Object o);
  int hashCode();

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

  boolean addAll(Collection<? extends E> c);
```

```
16    boolean containsAll(Collection<?> c);
17    boolean removeAll(Collection<?> c);
18    boolean retainAll(Collection<?> c);
19      :
20  }
```

Non-generic Methods

You might notice that, instead of `contains(E e)` and `remove(E e)`, the `Collection` interface uses `contains(Object o)` and `remove(Object o)`. This little inconsistency, however, is harmless. For instance, if you have a collection intended for circles only, adding a non-circle could be disastrous. Trying to remove an non-circle or checking for a non-circle, would just return false.

Java Collection Framework allows classes that implements an interface to throw an `UnsupportedOperationException` if the implementation decides not to implement one of the operations (but still need to have the method in the class).

The methods on Lines 9-10 should also be familiar. A collection can check if it is equal to another collection (which inevitably also a subclass of `Object`). As before, we will explain why we need `hashCode()` later. Just bear with it a little longer.

The method `toArray()` on Line 12 returns an array containing all the elements inside this collection. The second overloaded `toArray` method takes in an array of generic type `T`. If the collections fit in `a`, `a` is filled and returned. Else, it allocates a new array of type `T` and returned.

The second `toArray` method is a *generic method*. It is declared with `<T>` to indicate that the method can take any type `T`. When we call generic method, we do not have to pass in a type argument. Instead, the Java compiler infers the type from the arguments. If we call `toArray(new String[10)`, it would return a `String[]`, if we call `toArray(new Point[0])`, it would return a `Point[]` and so on. It is the caller resonsibility to pass in the right type, otherwise, an `ArrayStoreException` will be thrown.

The next group of methods operate on another collection. `addAll` add all the elements of collection `c` into the current collection; `containsAll` checks if all the elements of collection `c` are contained in the current collection; `removeAll` removes all elements from collection `c`, and finally, `retainsAll` remove all elements not in `c`.

What is more interesting about the methods is the type of `c`. In `containsAll`, for instance, the collection `c` has the type `Collection<?>`. `?` is known as wildcard type, or *unknown* type. This notation is used to denote the supertype of all parameterized interfaces created from `Collection<E>`.

In `addAll`, `c` is declared as `Collection<? extends E>`. The type parameter `<? extends E>` is an example of bounded type in generics. It means that the type argument is still unknown, but we know that it extends `E`. So, suppose I have a parameterized interface `Collection<Circle>` and `Circle extends PaintedCircle`, I can pass in a collection that has type `Collection<PaintedCircle>`.

Finally, let's get back to supertype of `Collection<E>`, `Iterable<E>`. The `Iterable<E>` interface provides only a single interface, `Iterator<E> iterator()`, which returns a generic interface called `Iterator<E>` over

the collection. An `Iterator` is another interface that allows us to go through all the elements in a `Collection<E>`. It has four method interfaces, three of which we will talk about today: `hasNext()`, which returns if there is a next element in the `Collection<E>`; `next()`, which returns the next element (with paramterized type `E`; and `remove()`, which removes the last returned element from the `Collection<E>`.

OK, so far I have talked about lots of methods but haven't showed any code. This is because Java Collection Framework does not provide a class that implements the `Collection<E>` directly. The documentation recommends that we implement the `Collection<E>` interface if we want a collection of objects that allows duplicates and does not care about the orders.

If you want to do so, however, it is likely more useful to inherit from the abstract class `AbstractCollection<E>` (which implements most of the basic methods of the interface) rather than implementing the interface `Collection<E>` directly.

Set and List

The `Set<E>` `List<E>` interfaces extend the `Collection<E>` class. `Set<E>` is meant for implementing a collection of objects that does not allow duplicates (but still does not care about order of elements), while `List<E>` is for implementing a collection of objects that allow duplicates, but the order of elements matters.

Mathematically, a `Collection<E>` is used to implement a bag, `Set<E>`, a set, and `List<E>`, a sequence.

Useful classes in Java collection that implements `List<E>` includes `ArrayList` and `LinkedList`, and useful classhes that implements `Set<E>` includes `HashSet`.

Comparator

The `List<E>` interface also specifies a `sort` method, with the following specification:

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

Remember at the end of Lecture 3 when we said there are "unpure" interfaces, that is interface that comes with implementation? This is one of them. The keyword `default` indicates that the interface `List<E>` comes with a default implementation of `sort` method. So a class that implements the interface needs not implement it again if they do not want to.

This method specification is also interesting and worth looking closer. It takes in an object `c` with generic interface `Comparator<? super E>`. Like `<? extends E>` that we have seen before, this is a *bounded* wildcard type. While `<? extends E>` is an unknown type upper bounded by `E`, `<? super E>` is an unknown type lower bounded by `E`. This means that we can pass in `E` or any supertype of E.

What does the `Comparator` interface do? We can specify how to compare two elements of a given type, by implementing a `compare()` method. `compare(o1,o2)` should return 0 if the two elements are equals, a negative integer if o1 is "less than" o2, and a positive integer otherwise.

Let's write `Comparator` class [1][#fn:2]:

```
class NameComparator implements Comparator<String> {
    public int compare(String s1, String s2) {
        return s1.compareTo(s2);
    }
}
```

In the above, we use the `compareTo` method provided by the `String` class to do the comparison. With the above, we can now sort the `names`:

```
names.sort(new NameComparator());
```

This would result in the sequence being changed to `<"Cersei","Gregor","Joffrey">`.

Map

The `Map<K,V>` interface is again generic, but this time, has two type parameters, `K` for the type of the key, and `V` for the type of the value.

The two most important methods for `Map` is `put` and `get`:

```
    V put(K key, V value);
    V get(Object k);
```

A useful class that implements `Map` interface is `HashMap`:

```
Map<String,Integer> population = new HashMap<String,Integer>();
population.put("Oldtown",500000);
population.put("Kings Landing",500000);
population.put("Lannisport",300000);
```

Later, if we want to lookup the value, we can:

```
population.get("Kings Landing");
```

Which Collection Class?

Java provides many collection classes, more than what we have time to go through. It is important to know which one to use to get the best performance out of them. For the few classes we have seen:

- Use `HashMap` if you want to keep a (key, value) pair for lookup later.
- Use `HashSet` if you have a collection of elements with no duplicates and order is not important.

- Use `ArrayList` if you have a collection of elements with possibly duplicates and order is important, and retriving a specific location is more important than removing elements from the list.
- Use `LinkedList` if you have a collection of elements with possibly duplicates and order is important, retriving a specific location is less important than removing elements from the list.

Further, if you want to check if a given object is contained in the list, then `ArrayList` and `LinkedList` are not good candidates. `HashSet`, on the other hand, can quickly check if an item is already contained in the set. There is unfortunately no standard collection class that supports fast `contain` and allow duplicates.

Sample Code

The following code was used to demonstrate sorting of an `ArrayList`.

```
1  import java.util.*;
2
3  class NameComparator implements Comparator<String> {
4    public int compare(String s1, String s2) {
5      // return (s1.compareTo(s2));
6      // return (s2.compareTo(s1));
7      return (s2.length() - s1.length());
8    }
9  }
10
11 class SortedList {
12   public static void main(String[] args) {
13     List<String> names = new ArrayList<String>();
14
15     names.add(0, "Joffrey");
16     names.add(1, "Cersei");
17     names.add(2, "Meryn");
18     names.add(3, "Walder");
19     names.add(4, "Gregor");
20     names.add(5, "Sandor");
21
22     System.out.println("Initial List");
23     for (String i: names) {
24       System.out.println(i);
25     }
26
27     names.sort(new NameComparator());
28
29     System.out.println("Sorted List");
30     for (String i: names) {
31       System.out.println(i);
32     }
33   }
34 }
```

---

1. Later in CS2030, you will see how we significantly reduce the verbosity of this code! But let's do it the hard way first.

Lecture 6: Nested Classes, Enum

Learning Outcomes

- Understand the need to override `hashCode` every time `equals` is overriden and how to use `Arrays`' `hashCode` method to compute a hash code
- Understand static vs non-static nested class, local class, and anonymous class and when to use / not to use one.
- Understand the rules about final / effectively final variable access of local class and anonymous class
- Aware of the limitation when declaring a new anonymous class
- Understand the concept of variable capture
- Be aware that `enum` is a special case of a `class` and share many features as a class -- not just constants can be defined.
- Understand how `enum` is expanded into a subclass of `Enum`
- Know that enum constants can have customized fields and methods.

Unfinished Business: Hash Code

But, what is important here is that, two keys (two objects, in general) which are the same (`equals()` returns `true`), must have the same `hashCode()` . Otherwise, `HashMap` would fail!

So it is important to ensure that if `o1.equals(o2)` , then `o1.hashCode() == o2.hashCode()` . Note that the reverse does not have to be true -- two objects with the same hash code does not have to be equals.

This property is also useful for implementing `equals()` . For a complex object, comparing every field for equality can be expensive. If we can compare the hash code first, we could filter out objects with different hash code (since they cannot be equal). We only need to compare field by field if the hash code is the same.

Calculating good hash code is an involved topic, and is best left to the expert (some of you might become expert in this), but for now, we can rely on the static `hashCode` methods in the `Arrays` class to help us. Let's add the following to `Point` :

```
1  public int hashCode() {
2    double[] a = new double[2];
3    a[0] = this.x;
4    a[1] = this.y;
```

```
    return Arrays.hashCode(a);
}
```

Revisit: Wildcards

A generic type can be instantiated with a wildcard `?` as type argument:

```
ArrayList<?> l = new ArrayList<Integer>();
```

You can think of `ArrayList<?>` as a short form for `ArrayList<? extends Object>`, which is different from `ArrayList<Object>`:

```
ArrayList<Object> l = new ArrayList<Integer>();
```

The above will result in an error.

The first time I showed you the wildcard type is in the signature of the `Collection<E>` class:

```
public boolean containsAll(Collection<?> c);
```

The implication of this declaration is that we can pass in collection of any object types. This provide lots of flexibility, but one could argue that there is probably too much flexibility! One could pass in a collection of `String` objects to check if a collection of `Integer` contains these `String` objects -- the code will say no, but it seems silly to allow this.

To understand why Java designer goes with the above, instead of a less silly:

```
public boolean containsAll(Collection<? extends E> c);
```

recall that generics is introduced only after Java 5. There are possibly a lot of legacy code that expects `containsAll` to take a collection of `Object` objects (since before generics is introduced, the only way to write generic class is to use `Object` like I showed you). To keep Java backward compatible, a little bit of silliness is worth it!

Nested Class

Nested classes are use to group logically relevant classes together. Typically, a nested class is tightly coupled with the container class, and would have no use outside of the container class. The nested classes are used to encapsulate information within class, for instance, when implementation of a class becomes too complex.

Take the `HashMap<K,V>` class for instance. `HashMap<K,V>` contains several nested classes, including `KeyIterator<K>`, `ValueIterator<V>` which implements a `Iterator<E>` interface for iterating through the

keys and the values in the map respectively, and an `Node<K,V>` class, which encapsulates a key-value pair in the map. These classes are declared `private`, since they are only used within the `LinkedList` class.

Nested class can be either static or non-static . Just like static fields and static methods, a *static nested class* is associated with the containing *class*, NOT *instance*. So, it can only access static fields and static methods of the containing class. A *non-static nested class*, on the other hand, can access all fields and methods of the containing class. A *non-static nested class* is also known as a *inner class*.

Note that a nested class can have read/write access even to the private fields and members of containing class. Thus, you should really have a nested class only if the nested class belongs to the same encapsulation. Otherwise, the containing class have a leaky abstraction barrier.

Local Class

We can declare a class within a function as well. One example is the `EventComparator` class in Lab 3. You might have felt silly to write a top-level class (in another file named `EventComparator.java`) just to compare two events, and you are right!

We can actually just define the `EventComparator` class when we need it, in the method that creates the event queue.

```
public EventQueue() {
  class EventComparator implements Comparator<Event> {
    public int compare(Event e1, Event e2) {
    return e1.compareTo(e2);
    }
  }
  events = new PriorityQueue<Event>(new EventComparator());
  :
}
```

Note that I am not putting two code snippets from different part of the code together, as I sometimes do. I am literally declaring the class inside the method where I initialize the `events` variable!

Classes declared inside a method (to be more precise, inside a block of code between `{` and `}`) is called a local class. Just like a local variable, a local class is scoped within the method. Like a nested class, a local class has access to the variables of the enclosing class.

Recall that when a method returns, all local variables of the methods are removed from the stack. But, an instance of that local class might still exist. For this reason, even though a local class can access the local variables in the enclosing method, the local class makes *a copy of local variables* inside itself. We say that a local class *captures* the local variables. It would be confusing if the copy of the variables inside the local class has a different value than the one outside. To avoid the confusion, Java only allow a local class to access variables that are explicitly declared `final` or implicitly final (or *effectively* final). An implicitly final variable is one that does not change after initialization.

Consider the following code:

```
boolean descendingOrder = false;
class EventComparator implements Comparator<Event> {
  public int compare(Event e1, Event e2) {
    if (descendingOrder) {
        return e2.compareTo(e1);
    } else {
        return e1.compareTo(e2);
    }
  }
}
descendingOrder = true;
events = new PriorityQueue<Event>(new EventComparator());
```

In what order will the event be sorted? Luckily, the designers of Java save us from such hair-pulling situation and disallow such code -- `descendingOrder` is not effectively final so the code will not compile.

I do not see a good use case for local class -- if you have information and behavior inside a block of code that is so complex that you need to encapsulate it within a local class, it is time to rethink your design!

What about the use case of `EventComparator` above? Well, if the class is short enough and is only used once, then it is a good use case for *anonymous class*.

Anonymous Class

An anonymous class is one where you declare a class and instantiate it in a single statement. We do not even have to give it a name!

```
events = new PriorityQueue<Event>(new Comparator<Event>() {
  public int compare(Event e1, Event e2) {
    return e1.compareTo(e2);
  }
});
```

The example above removes the need to declare just for the purpose of comparing two events. An anonymous class has the following format: `new X (arguments) { body }`, where:

- *X* is a class that the anonymous class extends or an interface that the anonymous class implements. X cannot be empty. This syntax also implies an anonymous class cannot extend another class and implement an interface at the same time. Furthermore, an anonymous class cannot implements more than one interface.
- *arguments* is the arguments that you want to pass into the constructor of the anonymous class. If the anonymous class is extending an interface, then there is no constructor, but we still need the two parenthesis `()`.
- *body* is the body of the class as per normal, except that we cannot have constructor for anonymous class.

The syntax might look overwhelming at the beginning, but we can also write it as:

```
1  Comparator<Event> cmp = new Comparator<Event>() {
2    public int compare(Event e1, Event e2) {
3      return e1.compareTo(e2);
4    }
5  }
6  events = new PriorityQueue<Event>(cmp);
```

Line 1 above looks just like what we do when we instantiate a class, except that we are instantiating an interface with a `{ .. }` body.

An anonymous class is just like a local class, it captures the variables of the enclosing scope as well -- the same rules to variable access as local class applies.

Enum

An `enum` is a special type of class in Java. Variable of an enum type can only be one of the predefined constants. Using enum has one advantage over the use of `int` for predefined constant -- it is type safe!

If we define the event type as enum, then we can write like this:

```
1  enum EventType {
2    CUSTOMER_ARRIVE,
3    CUSTOMER_DONE
4  }
```

and the field `eventType` in `Event` now has a type `EventType` instead of `int`:

```
1  class Event {
2    private double time;
3    private EventType eventType;
4  }
```

Trying to assign anything other than the two predefined event type to `eventType` would result in compilation error.

Each constant of an enum type is actually an instance of the enum class and is a field in the enum class declared with `public static final`.

Enum's Fields and Methods

Since enum in Java is a class, we can define constructors, methods, and fields in enums.

```
1  enum Color {
2    BLACK(0, 0, 0),
3    WHITE(1, 1, 1),
4    RED(1, 0, 0),
5    BLUE(0, 0, 1),
6    GREEN(0, 1, 0),
```

```
  YELLOW(1, 1, 0),
  PURPLE(1, 0, 1);

  private final double r;
  private final double g;
  private final double b;

  Color(double r, double g, double b) {
    this.r = r;
    this.g = g;
    this.b = b;
  }

  public double luminance() {
    return (0.2126 * r) + (0.7152 * g) + (0.0722 * b);
  }

  public String toString() {
    return "(" + r + ", " + g + ", " + b + ")";
  }
}
```

In the example above, we represent a color with its RGB component. Enum values should only constants, so `r`, `g`, `b` are declared as `final`. We have a method that computes the luminance (the "brightness") of a color, and a `toString()` method.

The enum values are now written as `BLACK(0, 0, 0)`, with arguments passed into constructor.

Custom Methods for Each Enum

Enum in Java is more powerful than the above -- we can define custom methods for each of the enum constant, by writing *constant-specific class body*. If we do this, then each constant becomes an anonymous class that extends the enclosing enum.

Consider the event type enum. We are going to diverge from your Labs 2 and 3 now [1] [#fn:1], so that I can bring you the following eample:

```
enum EventType {
  CUSTOMER_ARRIVE {
    // customer arrives at uniformly random interval [0, 0.5]
    double timeToNextEvent() {
      return rng.nextDouble()*0.5;
    }
  },
  CUSTOMER_DONE {
    // customer completes service at exponential random interval with mu = 1.5
    double timeToNextEvent() {
      return -Math.log(rng.nextDouble())/1.5;
    }
  };
  private static Random rng = new Random(1);
  abstract double timeToNextEvent();
}
```

In the code above, `EventType` is an abstract class -- `timeToNextEvent` is defined as `abstract` with no implementation. Each enum constant has its own implementation for calculation of time to next event.

Now, each event has its own method to generate the time to the next event of that type, and we can call

```
EventType.CUSTOMER_DONE.timeToNextEvent()
```

to get the time to the next event of that particular type!

The Class `Enum`

`enum` is a special type of class in Java. All `enum` inherits from the class `Enum` implicitly. Since `enum` is a class, we can extend `enum` from interfaces as per normal class. Unfortunately, `enum` cannot extend another class, since it already extends from `Enum`.

One implicitly declared method in `enum` is a static method:

```
public static E[] values();
```

We can call `EventType.values()` or `Color.values()` to return an array of event types or an array of colors. `E` is a type parameter, corresponding to the enum type (either `EventType`, `Color`, etc). To maintain flexibility and type safety, the class `Enum` which all enums inherit from has to be a generic class with `E` as a type paramter.

Considering `EventType`,

```
enum EventType {
  CUSTOMER_ARRIVE,
  CUSTOMER_DONE
}
```

is actually

```
public final class EventType extends Enum<EventType> {
  public static final EventType[] values { .. }
  public static EventType valueOf(String name) { .. }

  public static final EventType CUSTOMER_ARRIVE;
  public static final EventType CUSTOMER_DONE;
    :

  static {
    CUSTOMER_ARRIVE = new EventType();
    CUSTOMER_DONE = new EventType();
      :
  }
}
```

Even though we can't extend from `Enum` directly, Java wants to ensure that `E` must be a subclass of `Enum` (so that we can't do something non-sensical like `Enum<String>`. Furthermore, some methods from `Enum` (such as `compareTo()`) are inherited to the enum class, and these methods involved generic type `E`. To ensure that the generic type `E` actually inherits from `Enum<E>`, Java defines the class `Enum` to have bounded generic type `Enum<E extends Enum<E>>`.

Extra Note: This is a recursive generic construct *hack* to ensure that `E` is an enum type of `Enum<E>`.

The expansion of enum `EventType` to a class above also illustrates a few points:

- `enum` are *implicitly* finals. We cannot inherit from enum (those with constant-specifc body are exceptions).
- A class in Java can contains fields of the same class.
- The block marked by `static { .. }` are *static initializers*, they are called when the class is first used. They are the counterpart to constructors for objects, and are useful for non-trivial initialization of static fields in a class.

Enum-related Collections

Java Collection Frameworks provide two useful classes `EnumSet` and `EnumMap` -- they can be viewed as special cases of `HashSet` and `HashMap` respectively -- the only different is that we can only put enum values into `EnumSet` and enum-type keys into `EnumMap`.

---

1. This may or may not be the best way to solve Lab 3.