
COMP3021 - Final Review



Shing-Chi Cheung
Computer Science and Engineering
HKUST

Important Notes

- Dec 18 (Friday), 16:30-19:30am, Online over Canvas and Zoom. Open notes (hard copies only). 3 hours.
- Every topic covered in the course will be examined.
- Exam will mostly focus on concepts and principles.
- Your screen on desktop/laptop must be wholly occupied by Canvas
- Must enable your Zoom video camera on mobile phone during exam
- Video camera must show the side-view of you, your keyboard, mouse and the Canvas screen
- Bring along your student ID card
- Multiple choices: Each question can have multiple answers; no point received if any of the choices selected is incorrect
- Prepare white papers for sketch work for short questions

Examination Focuses

- Objects, classes, strings and inheritance
- Polymorphism, abstract classes and interfaces
- Exception Handling
- GUI and Event-driven programming (including inner classes and functional objects)
- Generics
- Multithreading
- Lambdas

Objects, classes, strings and inheritance

Comparison: Instance versus Static

Instance Variable and Method

- An **instance variable** belongs to a specific instance
- An **instance method** is invoked by an instance of its class

Static Variable, Method and Constant

- A **static variable** is shared by all the instances of its class
- A **static method** is not tied to a specific instance.
- A **constant** is a **static final variable** shared by all the instances of its class

Reference Variables and Default Initialization

- Instance and static variables are initialized with a **default** value 

```
class Student {  
    String name; // name has default value null  
    int age; // age has default value 0  
    boolean isScienceMajor; // isScienceMajor has default value false  
    char gender; // c has default value '\u0000'  
}
```

[DefaultValueTest.java](#)

NO default value is assigned to a local variable

```
public class NoDefault4Locals {  
    public static void main(String[] args) {  
        int x; // x does not have a default value  
        String y; // y does not have a default value  
        System.out.println("x is " + x);  
        System.out.println("y is " + y);  
    }  
}
```

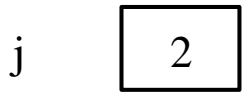
Compile error: variable not initialized

[NoDefault4Locals.java](#)

Copying Variables of Primitive Data Types and Object Types

`i = j; // primitive type assignment`

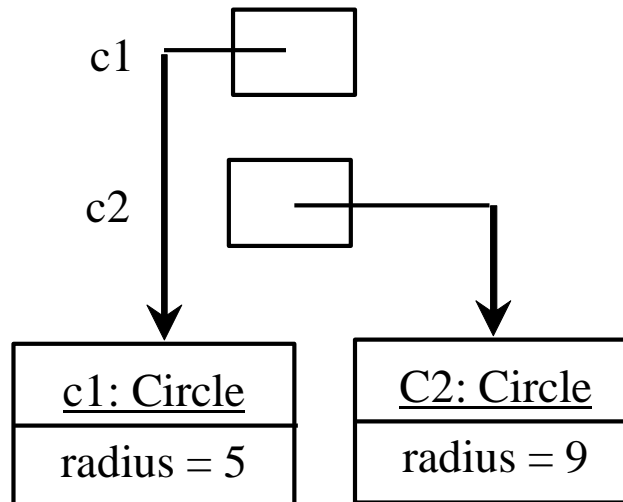
Before:



After:



Before:

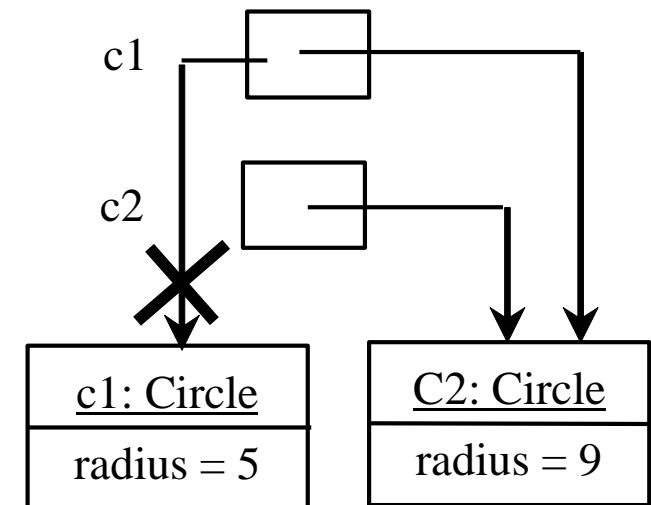


`Circle c1 = new Circle(5);`

`Circle c2 = new Circle(9);`

`c1 = c2; // obj type assignment`

After:




```
package p1;

public class C1 {
    public int x;
    int y;
    private int z;

    public void m1() {
    }
    void m2() {
    }
    private void m3() {
    }
}
```

```
package p1;

public class C2 {
    void aMethod() {
        C1 o = new C1();
        can access o.x;
        can access o.y;
        cannot access o.z;

        can invoke o.m1();
        can invoke o.m2();
        cannot invoke o.m3();
    }
}
```

```
package p2;

public class C3 {
    void aMethod() {
        C1 o = new C1();
        can access o.x;
        cannot access o.y;
        cannot access o.z;

        can invoke o.m1();
        cannot invoke o.m2();
        cannot invoke o.m3();
    }
}
```

```
package p1;

class C1 {
    ...
}
```

```
package p1;

public class C2 {
    can access C1
}
```

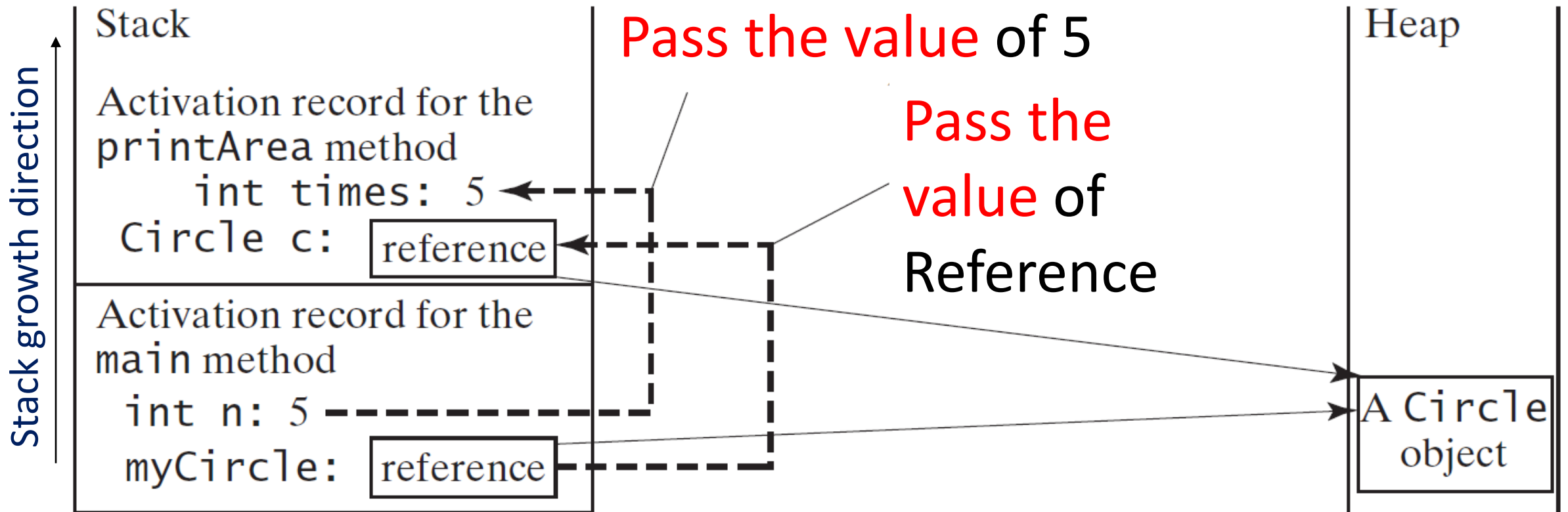
```
package p2;

public class C3 {
    cannot access C1;
    can access C2;
}
```

The private modifier restricts **access to within a class**, the default modifier restricts **access to within a package**, and the public modifier enables **unrestricted access**.

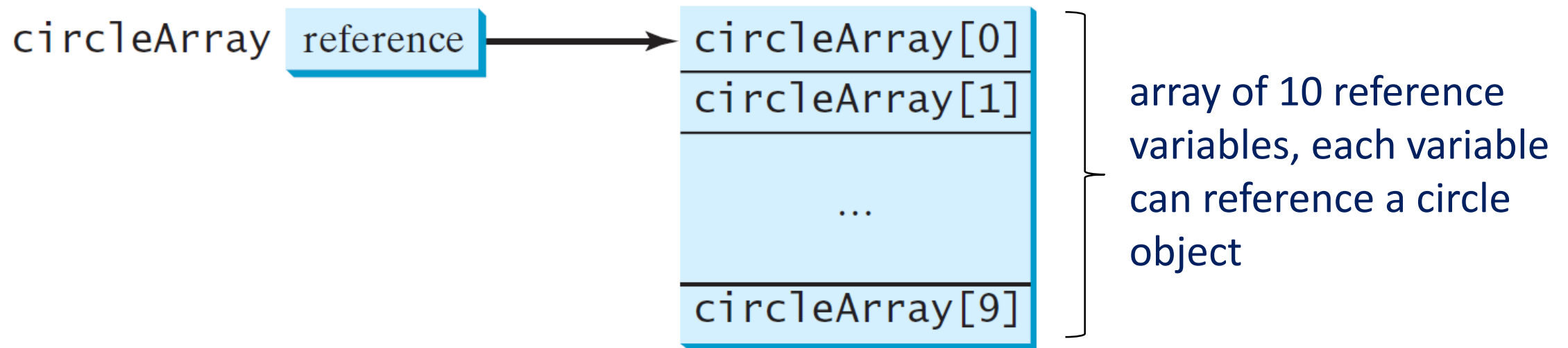


Passing **Values** to Method Parameters



Array of Objects

- `Circle[] circleArr = new Circle[10];`
- `Circle[]` denotes the type of Circle array
- An array of objects is actually an array of reference variables



Enhanced for-loop

Can be used with any class that implements the **Iterable<T>** interface, such as ArrayList



local-variable type inference

Old way

```
for (int i = 0; i < circleArr.length; i++)  
    sum += circleArr[i].getArea();
```

```
for (int i = 0; i < circleArr.length; i++)  
    System.out.printf("%-30f%-15f\n",  
        circleArr[i].getRadius(), circleArr[i].getArea());
```

New way since Java 10

```
for (var c: circleArr) // enhanced for loop  
    sum += c.getArea();
```

```
for (var c: circleArr) // enhanced for loop  
    System.out.printf("%-30f%-15f\n",  
        c.getRadius(), c.getArea());
```

*Note: We will introduce **forEach** - a powerful loop operator for streaming data at the end of this course. It supports lazy parallel stream processing.*

[TotalArea.java](#)

Immutable Objects and Classes

- **Immutable object**: An object whose content cannot be changed once it is created
- **Immutable class**: The class of an immutable object
- In our example, if the Circle class has only private variables and no set methods in the Circle class, it is **immutable** because radius is private and cannot be changed without a mutator (set) method
- In general, a class that contains only private variables and get methods is not necessarily **immutable**. For example, Student contains only private variables and get methods, but it is **mutable**

Comparison: Instance versus Static

Instance Variable and Method

- An **instance variable** belongs to a specific instance
- An **instance method** is invoked by an instance of its class

Static Variable, Method and Constant

- A **static variable** is shared by all the instances of its class
- A **static method** is not tied to a specific instance.
- A **constant** is a **static final variable** shared by all the instances of its class

Reference Variables and Default Initialization

- Instance and static variables are initialized with a **default** value 

```
class Student {  
    String name; // name has default value null  
    int age; // age has default value 0  
    boolean isScienceMajor; // isScienceMajor has default value false  
    char gender; // c has default value '\u0000'  
}
```

[DefaultValueTest.java](#)

NO default value is assigned to a local variable

```
public class NoDefault4Locals {  
    public static void main(String[] args) {  
        int x; // x does not have a default value  
        String y; // y does not have a default value  
        System.out.println("x is " + x);  
        System.out.println("y is " + y);  
    }  
}
```

Compile error: variable not initialized

[NoDefault4Locals.java](#)

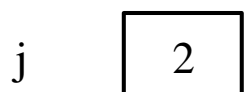
Copying Variables of Primitive Data Types and Object Types

`i = j; // primitive type assignment`

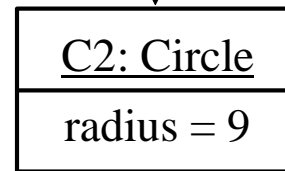
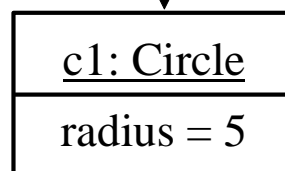
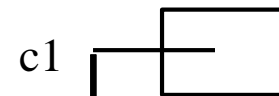
Before:



After:



Before:

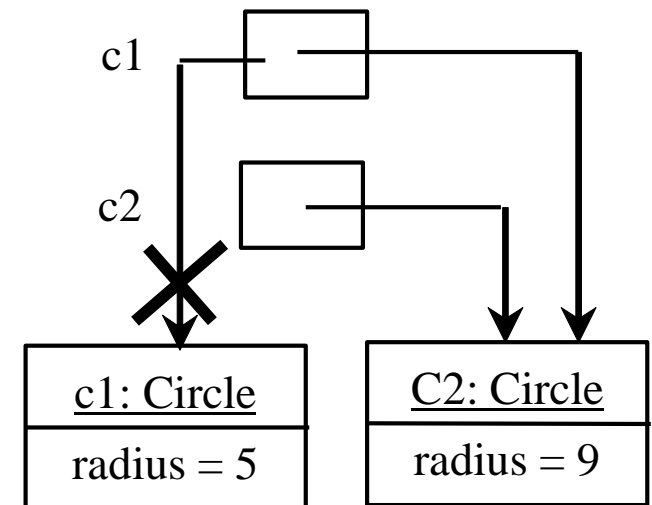


`Circle c1 = new Circle(5);`

`Circle c2 = new Circle(9);`

`c1 = c2; // obj type assignment`

After:



```
package p1;

public class C1 {
    public int x;
    int y;
    private int z;

    public void m1() {
    }
    void m2() {
    }
    private void m3() {
    }
}
```

```
package p1;

public class C2 {
    void aMethod() {
        C1 o = new C1();
        can access o.x;
        can access o.y;
        cannot access o.z;

        can invoke o.m1();
        can invoke o.m2();
        cannot invoke o.m3();
    }
}
```

```
package p2;

public class C3 {
    void aMethod() {
        C1 o = new C1();
        can access o.x;
        cannot access o.y;
        cannot access o.z;

        can invoke o.m1();
        cannot invoke o.m2();
        cannot invoke o.m3();
    }
}
```

```
package p1;

class C1 {
    ...
}
```

```
package p1;

public class C2 {
    can access C1
}
```

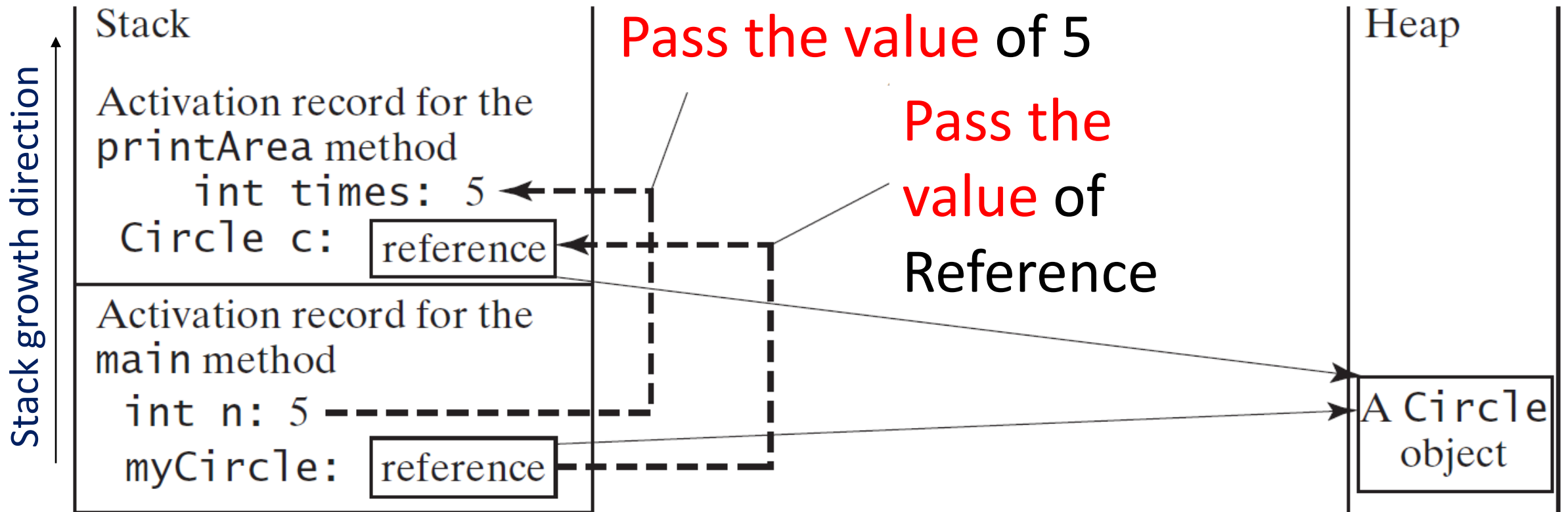
```
package p2;

public class C3 {
    cannot access C1;
    can access C2;
}
```

The private modifier restricts **access to within a class**, the default modifier restricts **access to within a package**, and the public modifier enables **unrestricted access**.

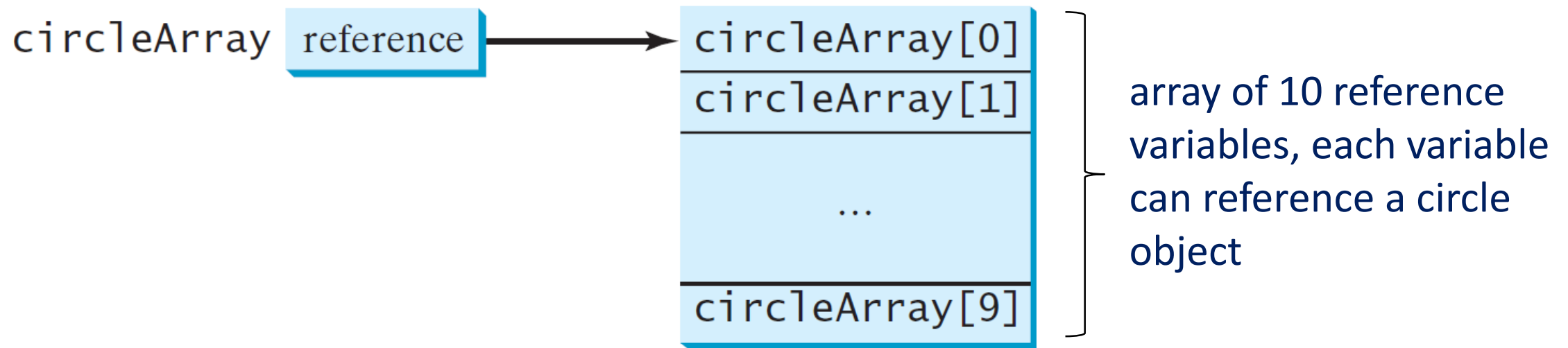


Passing Values to Method Parameters



Array of Objects

- `Circle[] circleArr = new Circle[10];`
- `Circle[]` denotes the type of Circle array
- An array of objects is actually an array of reference variables



Enhanced for-loop

Can be used with any class that implements the **Iterable<T>** interface, such as ArrayList



local-variable type inference

Old way

```
for (int i = 0; i < circleArr.length; i++)  
    sum += circleArr[i].getArea();
```

```
for (int i = 0; i < circleArr.length; i++)  
    System.out.printf("%-30f%-15f\n",  
        circleArr[i].getRadius(), circleArr[i].getArea());
```

New way since Java 10

```
for (var c: circleArr) // enhanced for loop  
    sum += c.getArea();
```

```
for (var c: circleArr) // enhanced for loop  
    System.out.printf("%-30f%-15f\n",  
        c.getRadius(), c.getArea());
```

*Note: We will introduce **forEach** - a powerful loop operator for streaming data at the end of this course. It supports lazy parallel stream processing.*

TotalArea.java

Immutable Objects and Classes

- **Immutable object**: An object whose content cannot be changed once it is created
- **Immutable class**: The class of an immutable object
- In our example, if the Circle class has only private variables and no set methods in the Circle class, it is **immutable** because radius is private and cannot be changed without a mutator (set) method
- In general, a class that contains only private variables and get methods is not necessarily **immutable**. For example, Student contains only private variables and get methods, but it is **mutable**

Are superclass constructors inherited?

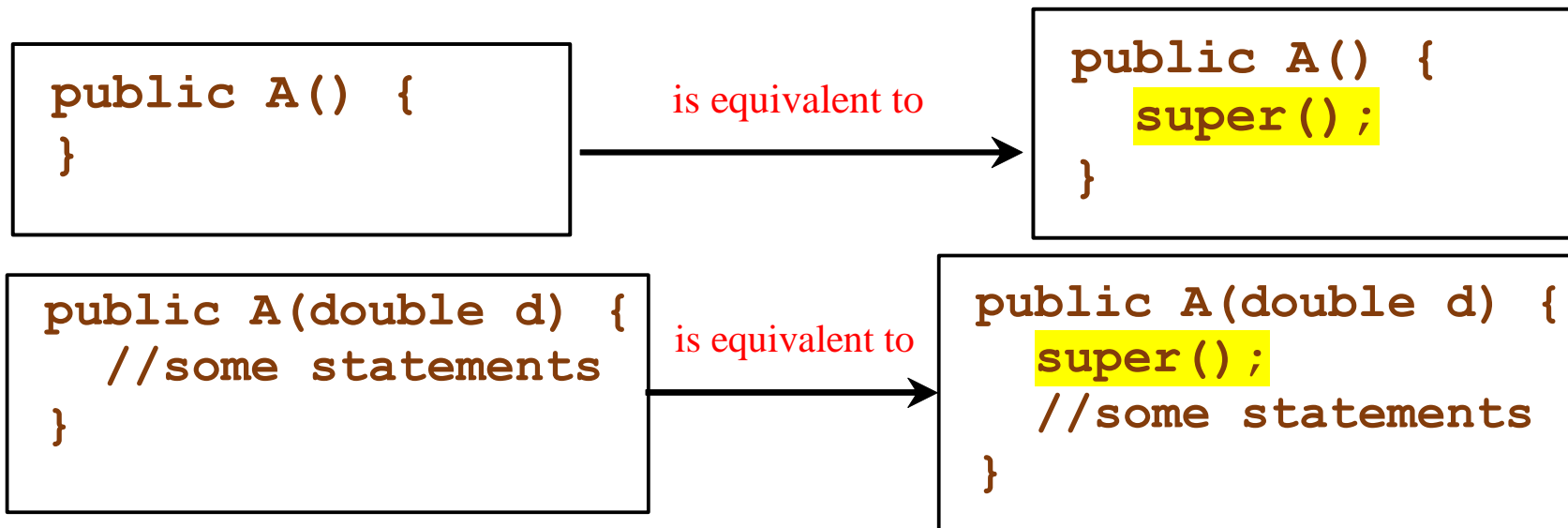
- Unlike properties and methods, a superclass constructors are **not inherited** by subclasses.
- Two ways to invoke a superclass constructor.
 - **Explicit invocation**: Superclass constructor can be invoked from a subclass constructor using the keyword **super**.
 - **Implicit invocation**: If the keyword **super** is not explicitly used in a subclass constructor, its superclass' **no-arg constructor** is automatically invoked.



[ConstructorInheritanceTest.java](#)

Superclass constructor is always invoked

- A constructor may invoke another overloaded constructor using keyword **this** or its superclass's constructor using keyword **super**.
- If none of them is invoked explicitly, Java compiler implicitly adds **super()** as **the first statement** in the constructor. For example,



The call to **super** must be the **first** statement in a constructor



```
public Boop() {  
    super();  
}
```



These are OK because a call to superclass

```
public Boop(int i) {  
    super();  
    size = i;  
}
```



constructor is explicitly coded in the first statement

```
public Boop() {  
}
```



These are OK because a call to `super()` is automatically inserted as the first statement

```
public Boop(int i) {  
    size = i;  
}
```



```
public Boop(int i) {  
    size = i;  
    super();  
}
```



Compilation error because a call to constructor is not the first statement

The call to **this** must be the **first** statement in a constructor



```
public Boop() {  
    this(0);  
}
```



These are OK
because a call to
another overloading
constructor is
explicitly coded in
the first statement

```
public Boop(int i) {  
    this();  
    size = i;  
}
```



```
public Boop() {  
}
```



```
public Boop(int i) {  
    size = i;  
}
```



These are OK

```
public Boop(int i) {  
    size = i;  
    this();  
}
```




Compilation error because
a call to another
overloading constructor is
not the first statement

First Statement Rule in Constructor



- The call to **super** must be the **first** statement in a constructor
 - The call to **this** must be the **first** statement in a constructor
- How is the rule related to constructor chaining?
 - Which constructor body must be **completely executed once before initializing any instance variables** of the new object? Why?
 - May a superclass constructor be called more than once? So?


NOTE

- An **instance method** of a superclass can be **overridden** **only if it is accessible**. 
- A **private** instance method **cannot be overridden** because it is **inaccessible** outside its own class.
- A **superclass private method** and a **subclass method** are completely **unrelated** even they share the same signature.

[OverrideTest.java](#)



NOTE

- Like an instance method, a **static method** can be **inherited**.
- However, a **static method** cannot be **overridden**. 
- If a superclass **static method** is redefined in a subclass, the method is **hidden**.

[StaticTest.java](#)



Liskov Substitution Principle (LSP)



Liskov Substitution Principle (LSP): *An instance of subtype must accept a method call if it is accepted by an instance of the supertype. Overriding rules are defined to observe LSP.*

When method f overrides method g

The class defining f is a subtype of the class defining g

f has the same parameter types and number of parameters as g

f's return type must be the same as or a subclass of g's return type

f cannot reduce the accessibility of g

Dynamic binding applies

We will see more rules that observe LSP when discussing other Java features.
Further reading: <https://reflectoring.io/lsp-explained/>

The **final** Modifier

- A **final** class cannot be extended:

```
final class Math { ... }
```

- A **final** variable is a constant:

```
final static double PI = 3.14159;
```

- A **final** method cannot be overridden by its subclasses.

```
final void myFinalMethod() { ... }
```

Instance Field Initialization Order with Superclass

When a subclass object is created, it will be initialized in the following order:


- Field initializers and instance initializer blocks of its superclass
- Constructor of its superclass
- Field initializers and instance initializer blocks of the current class
- Constructor of the current class

Account.java
SavingsAccount.java
Bank.java

Constructors can invoke methods.
Why need instance initializer blocks?



What happens when a class is loaded by JVM?

- All **static variables** are initialized to their default value (0, false, null)
- **Static field initializers** and **static initializer blocks** are executed in the order of their appearance
- Note that **static variables** are **initialized once** when its parent class is **loaded**. 

StaticFieldInitialization.java

Abstract classes and interfaces

Rules of Using Abstract Classes



Can an abstract class:

- ✗ ☐ contain private abstract methods
- ✓ ☐ be used to instantiate objects using the new operator
- ✗ ☐ be used to create an instance of itself
- ✓ ☐ have constructors // initialize instance variables
- ✓ ☐ have no abstract methods // provide base for extension or default impl
- ✓ ☐ have no methods // usage? better replaced by a tagging interface (discussed later)
- ✓ ☐ have a concrete superclass // applicable to most abstract classes
- ✓ ☐ have an abstract method overriding a concrete method
- ✓ ☐ be used as a type to declare variables

```
var gArray = new GeometricObject[10];
```

demo

demo



What is an interface? Why is it useful?

- In many ways, an **interface** is similar to an **abstract class**, but the intent of an interface is to specify **common features** or **capabilities** for objects. Examples of built-in interfaces are **Comparable**, **Serializable** and **Cloneable**
- Methods defined in an interface must be **abstract** unless they are explicitly declared to be **default** or **private**
- Methods defined in an interface **can only be overridden by public methods**

Rules of Using Interfaces



Can an interface

- ✗ ■ contain instance variables?
- ✗ ■ contain mutable static variables?
- ✗ ■ contain non-public immutable static variables?
- ✗ ■ have constructors?
- ✗ ■ have static initializer blocks?
- ✗ ■ contain abstract static methods?
- ✗ ■ contain non-abstract instance methods?
unless they are declared to be **default** (since Java 8) or **private** (since Java 9)



Omitting Modifiers in Interfaces

All **data fields** are *public final static* and all methods are *public abstract* in an interface. For this reason, these modifiers can be omitted, as shown below:



```
public interface T1 {  
    public static final int K = 1;  
  
    public abstract void p();  
}
```

equivalent to



```
public interface T1 {  
    int K = 1;  
  
    void p();  
}
```

A constant defined in an interface can be accessed using syntax `<interface name>.<CONSTANT NAME>` (e.g., `T1.K`).

IfcRules.java

Built-in Interface: Comparable<E>

- The **Comparable** is a built-in interface in the **java.lang** package.



```
public interface Comparable<E> {  
    public abstract int compareTo(@NotNull E o);  
}
```

*E is a generic type variable.
Assign E to the type of objects to
be compared; otherwise any
type of objects can be compared.*

- All built-in wrapper classes (e.g., Boolean) **override** the **toString()**, **equals(Object o)**, and **hashCode()** methods defined in **Object**.
- All built-in numeric wrapper classes (e.g., Integer) and the Character class implement the **Comparable** interface, and **implement** the **compareTo** method.

The Cloneable Interface



public interface Cloneable { }

empty body

- **Cloneable** is defined in the java.lang package as a **marker interface** whose body is empty.
- A **marker interface** has an empty body. It denotes a specific capability
- An instance of the class that implements Cloneable can be cloned
Example: `class CloneableCircle extends Circle implements Cloneable { ... }`
- **clone()** is a **protected instance method** defined in the **Object** class for cloning.
- *Note: Java uses the clone() method instead of copy constructors*



```
java.lang.Object  
protected Object clone()  
throws CloneNotSupportedException
```



Summary - Rules of Using Interfaces



- All **interface variables** are **public, static and final**
- All **interface methods** are **public and abstract**. They cannot be declared static or final. Why?
- A **class** can **extend one superclass** and **implement multiple interfaces**
- An **interface** can **extend multiple interfaces** but it **cannot extend any classes**
- An **interface** can be used as a **type**



Interfaces vs. Abstract Classes



Data fields of an interface must be constants; an abstract class can have all types of data fields.

Methods of an interface must be abstract; an abstract class can have non-abstract methods.

	<i>Variables</i>	<i>Constructors</i>	<i>Methods</i>
Abstract class	No restrictions.	Constructors are invoked by subclasses through constructor chaining. An abstract class cannot be instantiated using the new operator.	No restrictions.
Interface	All variables must be public static final .	No constructors. An interface cannot be instantiated using the new operator.	All methods must be public abstract instance methods (unless they are explicitly declared to be private or default)

Exception

Checked Exceptions vs. Unchecked Exceptions

- RuntimeException, Error and their subclasses are known as *unchecked exceptions*. All other exceptions are known as *checked exceptions*, meaning that the compiler forces the programmer to check and deal with the exceptions.

Declaring Exceptions

Every method must state the types of checked exceptions it might throw. This is known as *declaring exceptions*.

```
public void myMethod()  
    throws IOException
```

```
public void myMethod()  
    throws IOException, OtherException
```

Throwing Exceptions

When a program detects an error, the program can create an instance of an appropriate exception type and throw it. This is known as *throwing an exception*. Here is an example,

```
throw new TheException();
```

```
TheException ex = new TheException();  
throw ex;
```

Catching Exceptions

```
try {  
    statements;    // Statements that may throw exceptions  
}  
catch (Exception1 exVar1) {  
    handler for exception1;  
}  
catch (Exception2 exVar2) {  
    handler for exception2;  
}  
...  
catch (ExceptionN exVar3) {  
    handler for exceptionN;  
}
```

Rethrowing Exceptions

```
try {  
    statements;  
}  
catch (TheException ex) {  
    perform operations before exits;  
    throw ex;    // throw new TheException();  
}
```


Be Careful with Multiple Exceptions

Note that the following will produce a compiling error. Why?

```
try {...}
```

```
catch (Exception e3) {...}
```

```
catch (ArithmeticException e1) {...}
```

```
catch (IOException e2) {...}
```

MultipleException

Multi-threading

run()

- The run() methods in a task class specifies how to perform the task. It's automatically invoked by JVM when a thread is started.
- You should not invoke it: doing so merely executes this method in the same thread; no new thread is started.

Creating Tasks and Threads

`java.lang.Runnable`  `TaskClass`

```
// Custom task class
public class TaskClass implements Runnable {
    ...
    public TaskClass(...) {
        ...
    }

    // Implement the run method in Runnable
    public void run() {
        // Tell system how to run custom thread
        ...
    }
    ...
}
```

(a)

```
// Client class
public class Client {
    ...
    public void someMethod() {
        ...
        // Create an instance of TaskClass
        TaskClass task = new TaskClass(...);

        // Create a thread
        Thread thread = new Thread(task);

        // Start a thread
        thread.start();
        ...
    }
    ...
}
```

(b)

Thread States

A thread can be in one of the following four major states

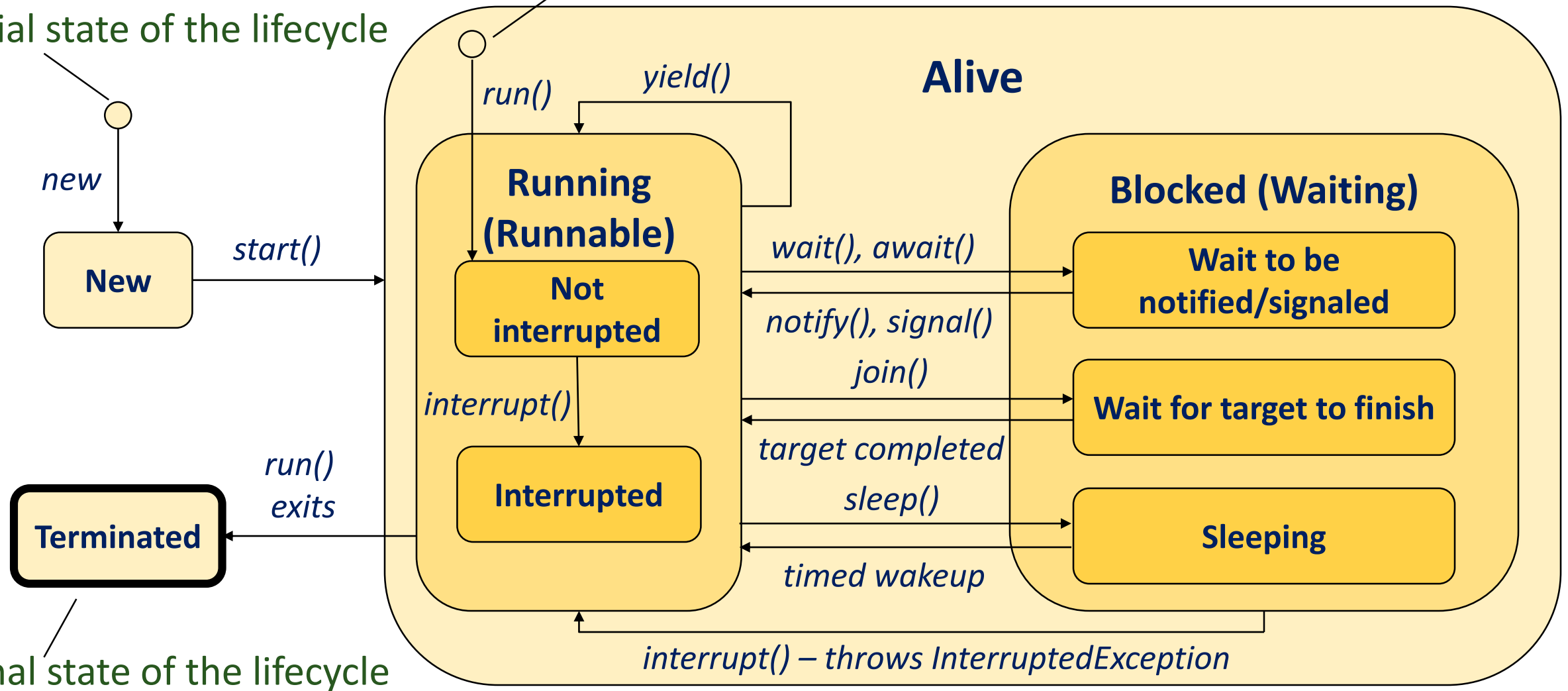
- ❑ **New**: thread is created but not yet started
- ❑ **Running**: thread is either being executed or to be scheduled for execution
- ❑ **Blocked**: thread is either blocked by monitor locks (to be discussed soon) or awaiting some events for a specified waiting time
- ❑ **Terminated**: thread exited

Thread Lifecycle



initial state of the lifecycle

initial state when a thread is alive; it is a transient state



The Static yield() Method

- We can use the yield() method to release time for other threads. For example, we can modify the code in TaskThreadDemo.java as follows:
- Every time a number is printed, the print100 thread is yielded. So, the numbers are printed after the characters.

```
public void run() {  
    for (int i = 1; i <= lastNum; i++) {  
        System.out.print(" " + i);  
        Thread.yield();  
    }  
}
```

The Static sleep(milliseconds) Method

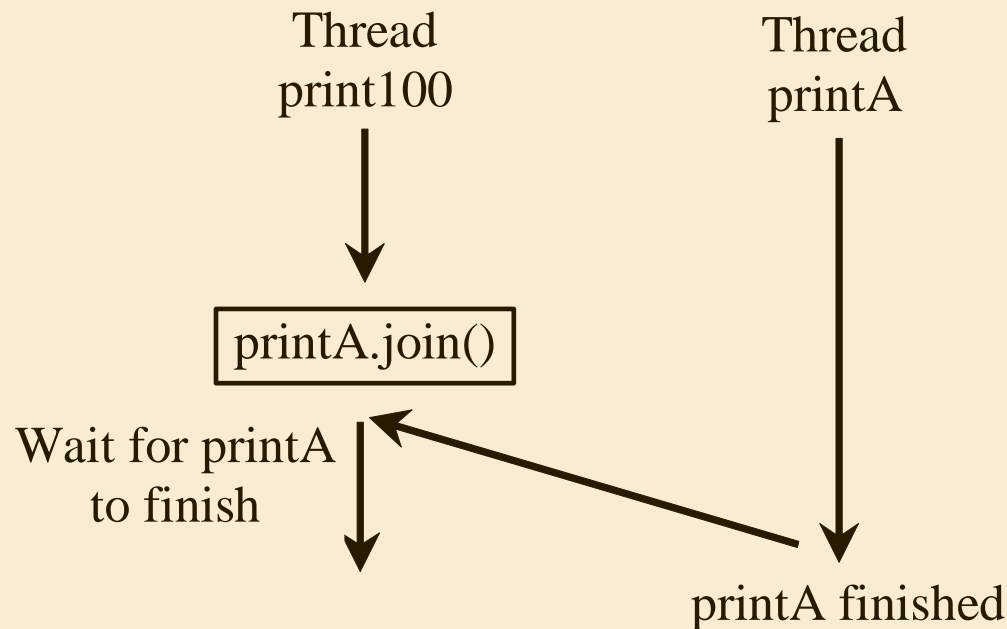
- The sleep(long mills) method puts the thread to sleep for the specified time in milliseconds. For example, we can modify the code in TaskThreadDemo.java as follows:
- Every time a number (≥ 50) is printed, the print100 thread is put to sleep for 1 millisecond.

```
public void run() {  
    for (int i = 1; i <= lastNum; i++) {  
        System.out.print(" " + i);  
        try {  
            if (i >= 50) Thread.sleep(1);  
        }  
        catch (InterruptedException ex) {  
        }  
    }  
}
```


The join() Method

- We can use the join() method to force one thread to wait for another thread to finish. For example, we can modify the code in TaskThreadDemo.java as follows:

```
public void run() {  
    Thread thread4 = new Thread(  
        new PrintChar('c', 40));  
    thread4.start();  
    try {  
        for (int i = 1; i <= lastNum; i++) {  
            System.out.print(" " + i);  
            if (i == 50) thread4.join();  
        }  
    } catch (InterruptedException ex) {  
    }  
}
```



The numbers after 50 are printed after thread printA is finished.

isAlive(), interrupt(), and isInterrupted()

- The isAlive() method returns true if the thread is in the Ready, Blocked, or Running state; it returns false if the thread is new and has not started or if it is finished.
- The interrupt() method interrupts a thread in the following way: the thread's interrupted flag is set if it is currently in the Ready or Running state. Otherwise, the thread is currently blocked; it awakes, enters the Ready state, and throws java.io.InterruptedIOException.
- The isInterrupted() method tests whether the thread is interrupted.

Thread Synchronization

Step	balance	thread[i]	thread[j]
1	0	<code>newBalance = bank.getBalance()+1;</code>	
2	0		<code>newBalance = bank.getBalance()+1;</code>
3	1	<code>bank.setBalance(newBalance);</code>	
4	1		<code>bank.setBalance(newBalance);</code>

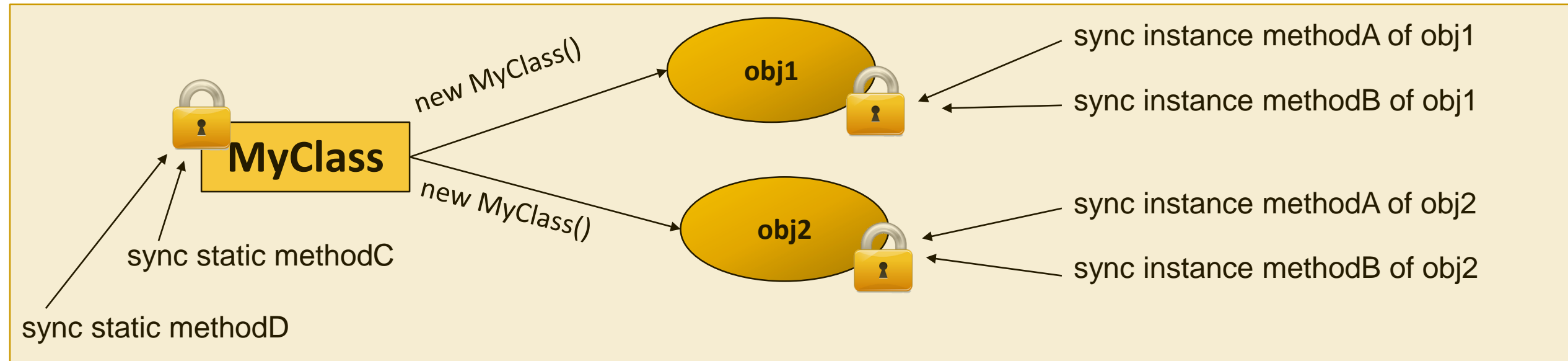
- A shared resource may be corrupted if it is accessed simultaneously by multiple threads. For example, two unsynchronized threads accessing the same bank account may cause conflict.

The **synchronized** keyword

- To avoid race conditions, more than one thread must be prevented from simultaneously entering certain part of the program, known as **critical region**. The **critical region** is the entire deposit method. We can use the **synchronized** keyword to synchronize the method so that only one thread can access the method at a time. There are several ways to correct the previous race problem. One approach is to make Account thread-safe by adding the **synchronized** keyword in the deposit method:

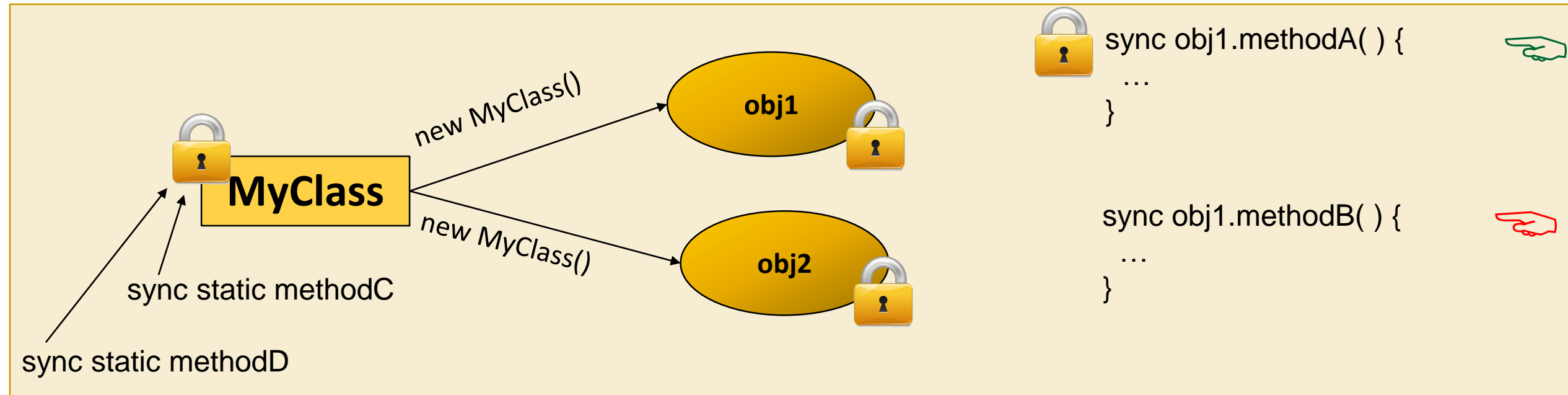
```
public synchronized void deposit(double amount) { ... }
```

Synchronizing Instance Methods and Static Methods



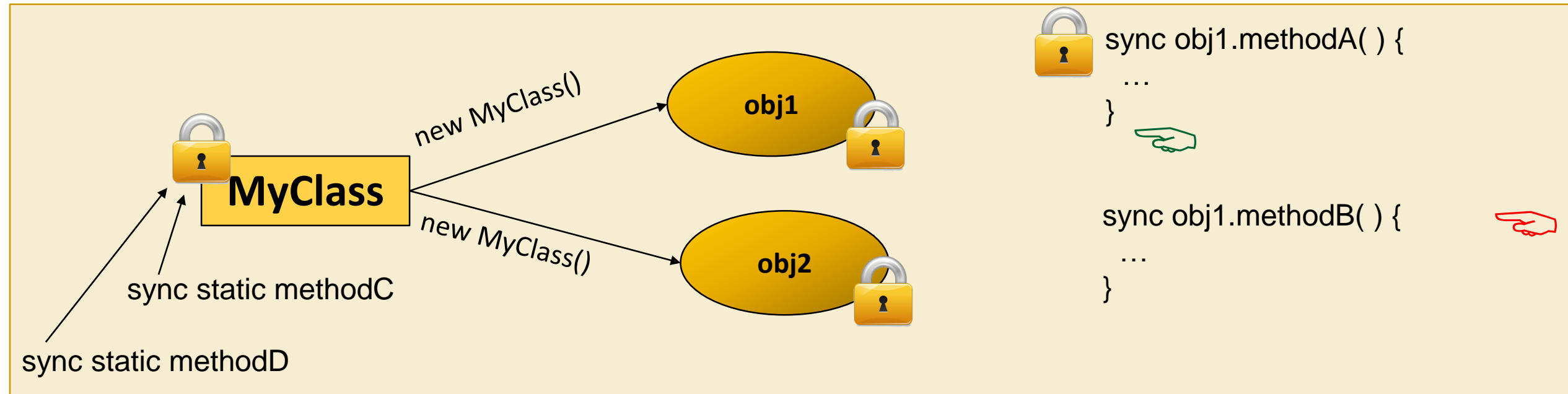
- A synchronized method acquires a lock before it executes.
 - In the case of an instance method, the lock is on the object for which the method was invoked.
 - In the case of a static method, the lock is on the class.

Synchronizing Instance Methods and Static Methods



- If one thread invokes a synchronized instance method (respectively, static method) on an object, the lock of that object (respectively, class) is acquired first, then the method is executed, and finally the lock is released. Another thread invoking the same method of that object (respectively, class) is blocked until the lock is released.

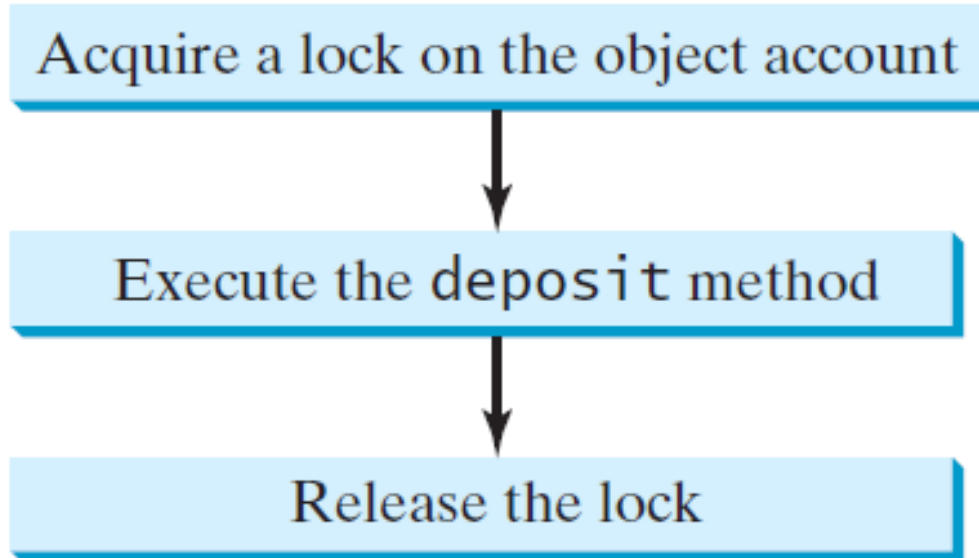
Synchronizing Instance Methods and Static Methods



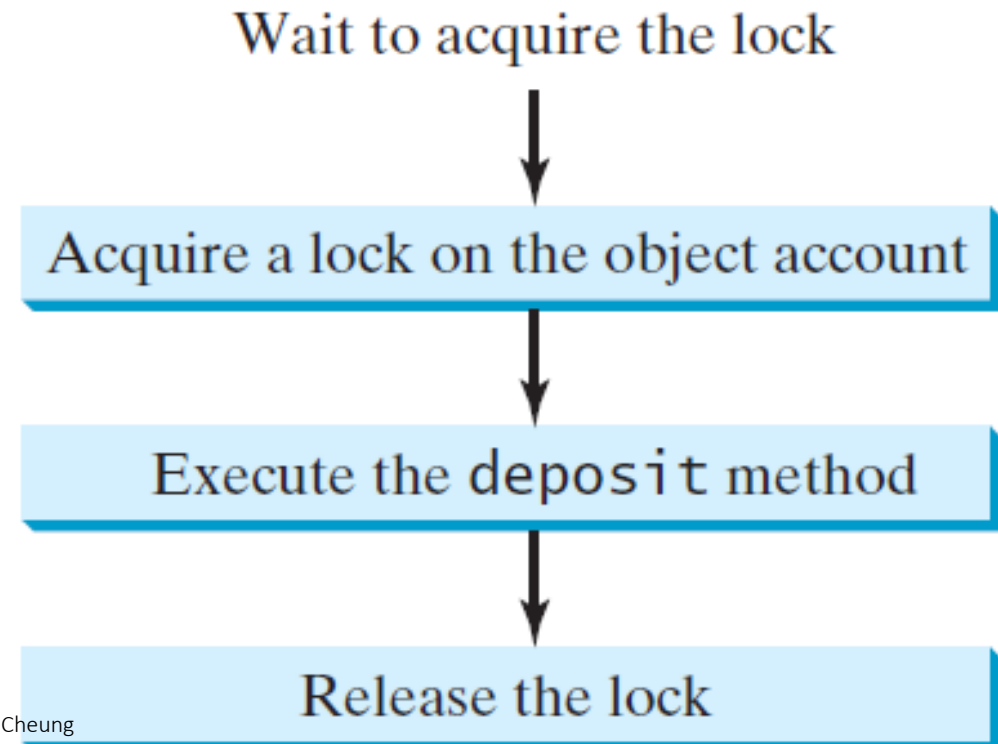
- If one thread invokes a synchronized instance method (respectively, static method) on an object, the lock of that object (respectively, class) is acquired first, then the method is executed, and finally the lock is released. Another thread invoking the same method of that object (respectively, class) is blocked until the lock is released.

Synchronizing Tasks

Task 1



Task 2



Synchronizing Statements

```
synchronized (expr) {  
    statements;  
}
```

- Invoking a synchronized instance method of an object acquires a lock on the object, and invoking a synchronized static method of a class acquires a lock on the class.
- A synchronized statement can be used to acquire a lock on any object, not just this object, when executing a block of the code in a method. This block is referred to as a **synchronized block**.

Synchronizing Statements



```
synchronized (expr) {  
    statements;  
}
```

- The expression `expr` must evaluate to an object reference. If the object is already locked by another thread, the thread is blocked until the lock is released.
- When a lock is obtained on the object, the statements in the synchronized block are executed, and then the lock is released.

Thread Synchronization using Synchronized Statements

```
synchronized (expr) { // expr must be evaluated to an object/class reference  
    statements;  
}
```

identifies a lock of
an object or a class



- A synchronized statement can be used to acquire a lock on a specific object, not limited to **this** object, when executing a block of the code in a method. This block is referred to as a **synchronized block**

- `synchronized (this) { ... }` `// synchronized using the current instance`
- `synchronized (new Account()) { ... }` `// synchronized using a new instance`
- `synchronized (Account.class) { ... }` `// synchronized using a class`

Thread Synchronization using Synchronized Statements

```
synchronized (expr) { // expr must be evaluated to object/class reference  
    statements;  
}
```

identifies a lock of
an object or a class



- The expression **expr** must be evaluated to an **object reference** or a **class reference**. If the object or class is already locked by another thread, the thread is blocked until the lock is successfully acquired
 - When the lock is acquired, the statements in the **synchronized block** are executed, and then the lock is released after these statements have been executed.
- [AccountWithSyncBlock.java](#)

Synchronized Statements vs. Synchronized Methods

- Any **synchronized instance method** can be converted into an instance method with a **synchronized block**



```
public synchronized void xMethod() {  
    // method body  
}
```

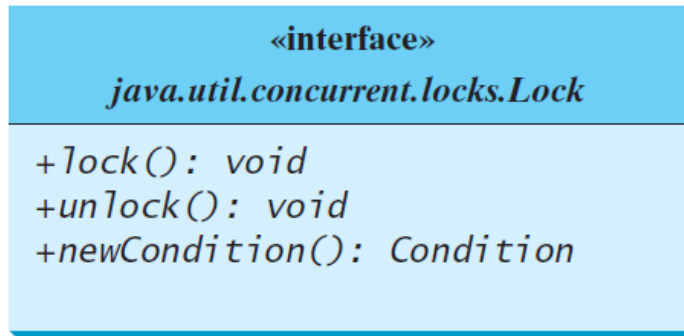
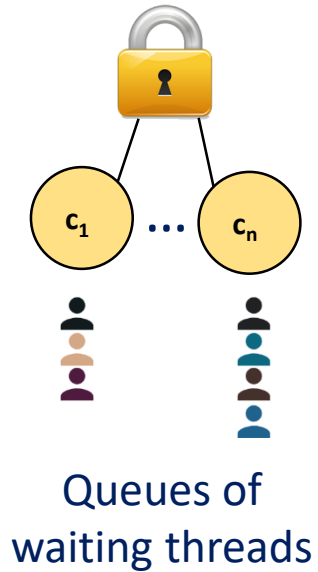


```
public void xMethod() {  
    synchronized (this) {  
        // method body  
    }  
}
```

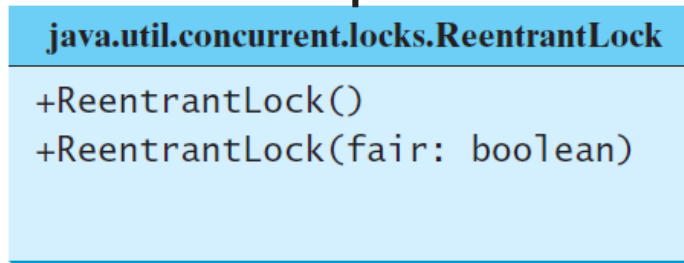
Q: Why synchronized statements?

A: **Synchronized statements** enable us to synchronize threads at **statement granularity** instead of **method granularity**

Thread Synchronization Using Explicit Locks



Acquires the lock.
Releases the lock.
Returns a new `Condition` instance that is bound to this `Lock` instance.

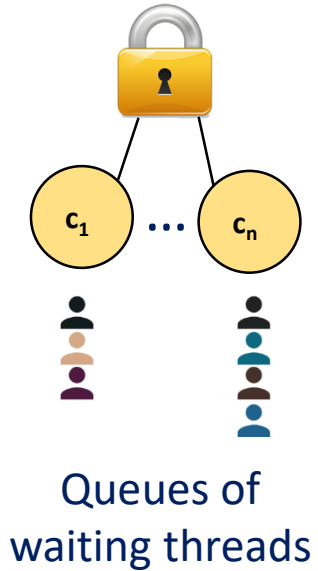


Same as `ReentrantLock(false)`.
Creates a lock with the given fairness policy. When the fairness is true, the longest-waiting thread will get the lock. Otherwise, there is no particular access order.

Need to
call these
methods
explicitly

- Besides using monitor-based locks, we can synchronize threads by explicit locks
- An explicit lock is an instance of the **Lock** interface, which declares the methods to acquire and release locks. A lock may use the **newCondition()** method to **create** any number of **Condition objects**, which can be used for thread communications

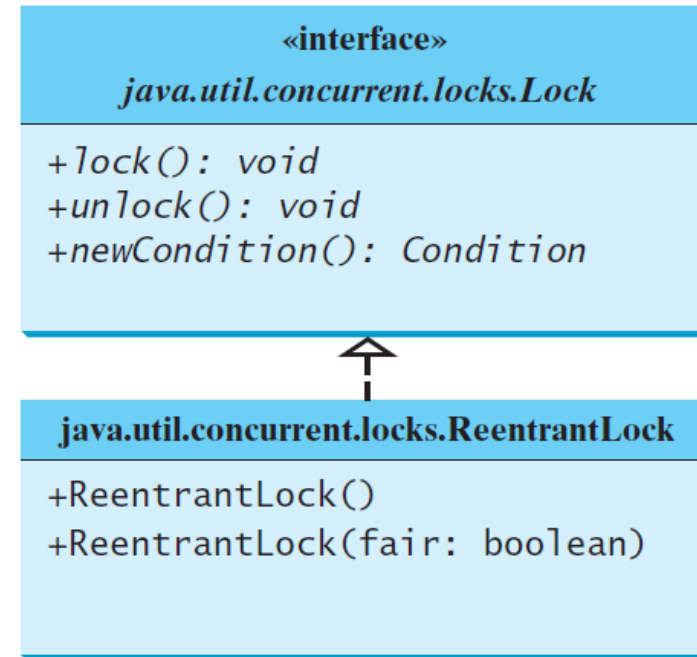
Thread Synchronization Using Explicit Locks



Example: A lock of a n-slot buffer

- c_1 : buffer is non-empty
 - consumer tasks queue for c_1
- c_2 : buffer is not full
 - producer tasks queue for c_2

In contrast: implicit lock – one lock and one condition per object (more restrictive)



Need to call these methods explicitly

- Besides using monitor-based locks, we can synchronize threads by explicit locks
- An explicit lock is an instance of the **Lock** interface, which declares the methods to acquire and release locks. A lock may use the **newCondition()** method to **create** any number of **Condition objects**, which can be used for thread communications

Cooperation Among Threads



«interface»

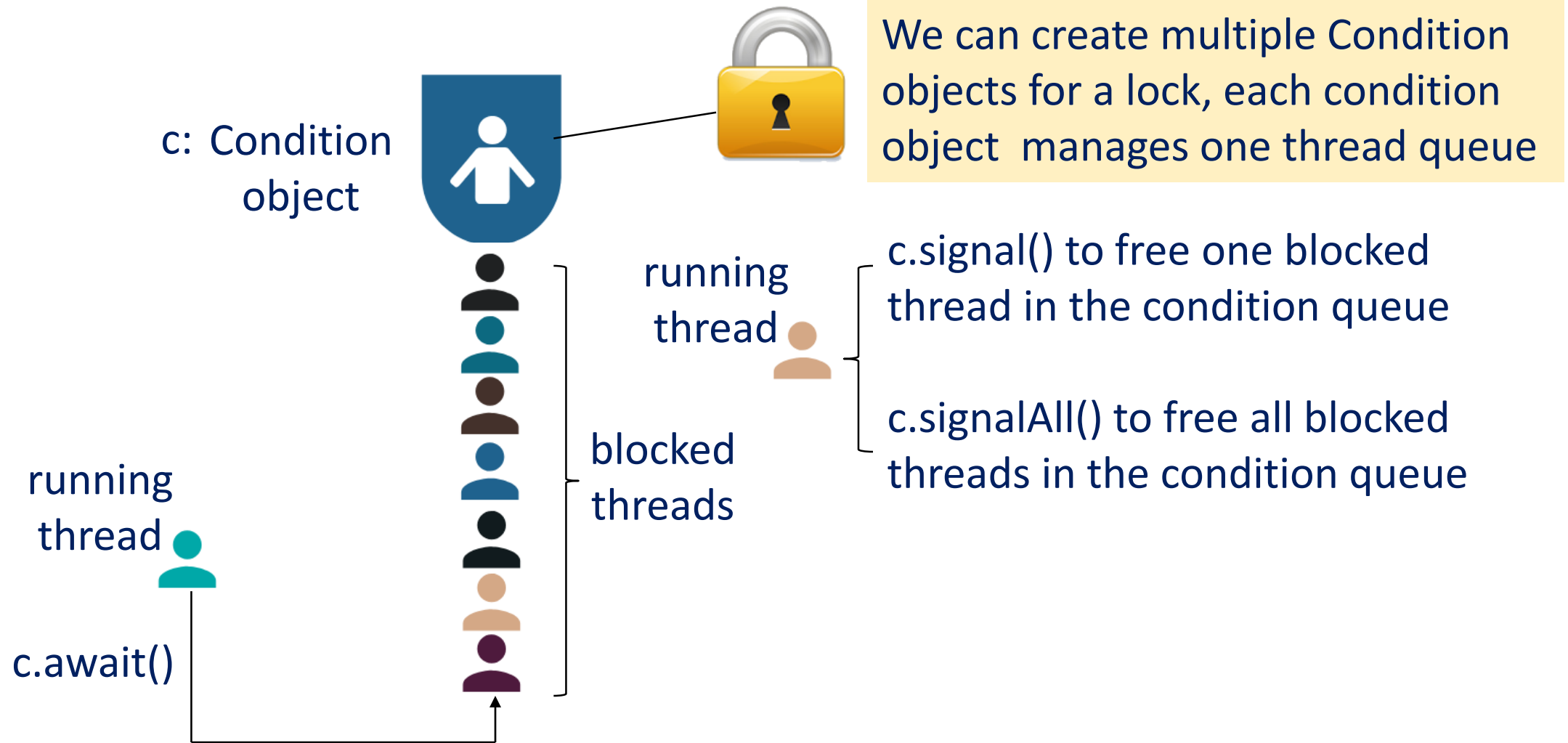
java.util.concurrent.Condition

```
+await(): void  
+signal(): void  
+signalAll(): Condition
```

Causes the current thread to wait until the condition is signaled.
Wakes up one waiting thread.
Wakes up all waiting threads.

- The conditions facilitate communications among threads. A thread can specify what to do under each condition
- Conditions are objects created by invoking the `newCondition()` method on a Lock object. Once a condition is created, we use its `await()`, `signal()`, and `signalAll()` methods for thread communications
- `await()` causes the current thread to wait until the condition is signaled; `signal()` wakes up one waiting thread, and `signalAll()` wakes up all waiting threads

Condition – await(), signal(), signalAll()



Example: Thread Cooperation



Suppose we launch two threads: one deposits to an account, and the other withdraws from the same account.

The withdraw thread awaits if the amount to be withdrawn is over the current account balance. Whenever the deposit thread adds new fund to the account, it signals the withdraw thread to resume. If the amount is still not enough for the withdrawal, the withdraw thread continues to await more fund in the account.

Assume the initial balance is 0 and the amount to deposit and to withdraw is randomly generated.

```
Command Prompt
C:\book>java ThreadCooperation
Thread 1          Thread 2          Balance
Deposit 7
Deposit 1
Deposit 10
Withdraw 9
Withdraw 4
Withdraw 3
Deposit 9
Withdraw 5
Withdraw 2
Deposit 3
Balance
7
8
18
9
5
2
11
6
4
7
```

Cooperation Among Threads

Withdraw Task

```
lock.lock();
```

```
while (balance < withdrawAmount)  
    newDeposit.await();
```

```
balance -= withdrawAmount
```

```
lock.unlock();
```

```
Condition newDeposit  
= lock.newCondition();
```

withdraw
thread

newDeposit.await()

blocked
threads



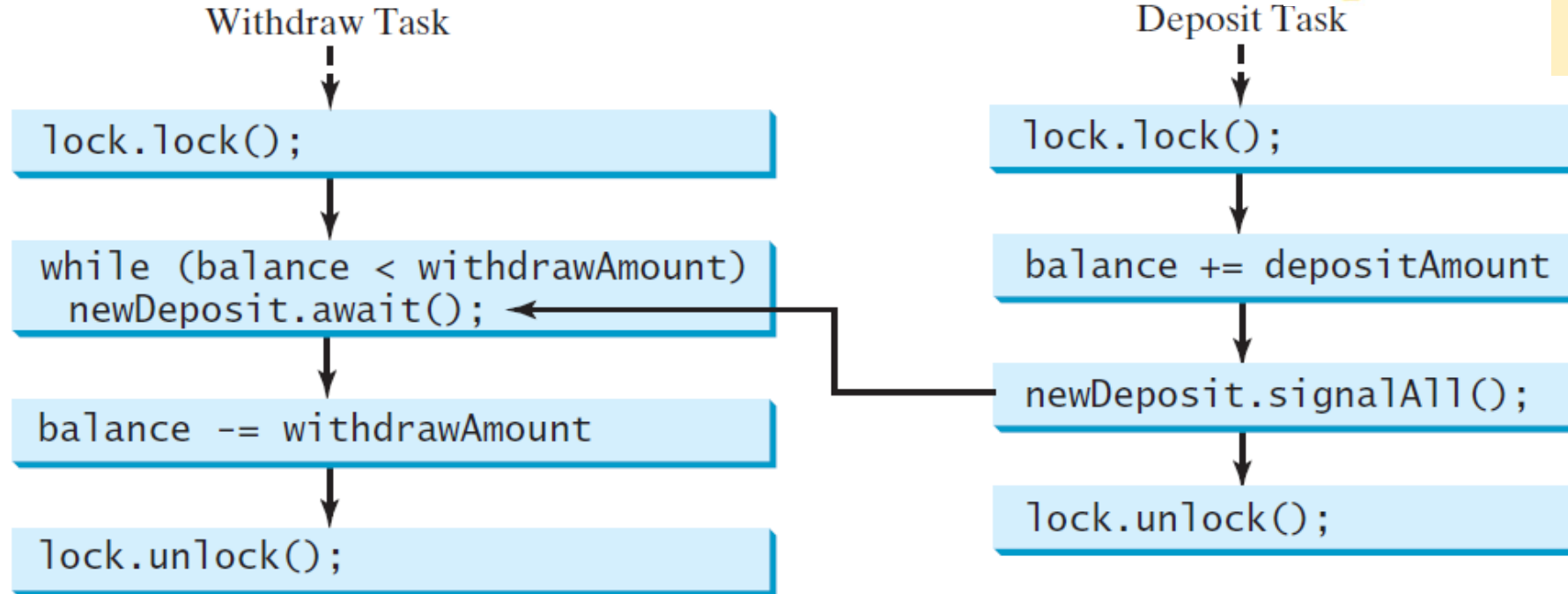
- Use a lock with a condition: newDeposit (to indicate new deposit added to the account)
- If the balance is less than the amount to be withdrawn, the withdraw task will wait for the newDeposit condition. When the deposit task adds money to the account, the task signals the waiting withdraw task to try again

ThreadCooperation.java

Cooperation Among Threads



Condition **newDeposit**
= **lock.newCondition()**;



- Use a lock with a condition: newDeposit (to indicate new deposit added to the account)
- If the balance is less than the amount to be withdrawn, the withdraw task will wait for the newDeposit condition. When the deposit task adds money to the account, the task signals the waiting withdraw task to try again

ThreadCooperation.java

wait(), notify(), and notifyAll()

- Use the `wait()`, `notify()`, and `notifyAll()` methods to facilitate communication among threads.
- The `wait()`, `notify()`, and `notifyAll()` methods must be called in a **synchronized method** or a **synchronized block** on the calling object of these methods. Otherwise, an **IllegalMonitorStateException** would occur
- The `wait()` method lets the thread wait until some condition occurs and immediately release the monitor lock of the object that owns the `wait()` method
- When the condition occurs, we use the `notify()` or `notifyAll()` methods to notify the waiting threads to resume normal execution. The `notifyAll()` method wakes up **all** waiting threads, while `notify()` picks up only **one** thread from a waiting queue
- When a waiting thread wakes up, it re-acquires the monitor lock that it has released

Generics

Generics



Shing-Chi Cheung
Computer Science and Engineering
HKUST

Terminology



```
class ArrayList<E> {  
    ...  
}
```

```
void func {  
    var cities = new ArrayList<String>();  
    ...  
}
```

- ArrayList<E> is a **generic class** or **generic type**
- E is a **type parameter** or **type variable**
- ArrayList<String> is a **parameterized type** of ArrayList<E>
- String is an **actual type argument** for E

Usage



No need to
put down
E o = list...




```
public class GenericStack<E> {  
    private ArrayList<E> list = new ArrayList<>();  
    public int getSize() { return list.size(); }  
    public E peek() { return list.get(getSize() - 1); }  
    public void push(E o) { list.add(o); }  
    public E pop() {  
        var o = list.get(getSize() - 1); // since Java 10  
        list.remove(getSize() - 1);  
        return o;  
    }  
    public boolean isEmpty() { return list.isEmpty(); }  
    @Override  
    public String toString() { return "stack: " + list.toString(); }  
}
```

With the support of
local variable type
inference, we do not
need to use type
parameters to declare
local variables
➔ more maintainable
code


No Casting Needed if Type Parameter is binded

- Casting is not needed to retrieve a value from an arraylist created with an actual argument type (say String)
- The String type of s will be inferred

```
var list = new ArrayList<String>();  
list.add("Red");  
list.add("White");  
var s = list.get(0); // No casting is needed
```



```
var list = new ArrayList();  
list.add("Red");  
list.add(1);  
String s = (String) list.get(0); // Casting is needed
```



Raw Type is Unsafe - 1

```
interface Comparable<E> {  
    public int compareTo(E o);  
}
```

```
public class ComparableCircleWithoutGeneric extends Circle  
    implements Comparable {
```

...

```
public int compareTo(Object o) {  
    double diff = this.getArea() - ((Circle) o).getArea();  
    if (diff > 0)  
        return 1;  
    else if (diff < 0)  
        return -1;  
    else  
        return 0;  
}
```

// triggers a runtime error if o is not passed a circle
new ComparableCircleWithoutGeneric(1).compareTo("RED");

ComparableCircleWithoutGeneric.java

Make it Safe with Generics



```
interface Comparable<E> {  
    public int compareTo(E o);  
}
```

```
public class ComparableCircleWithGeneric extends Circle  
    implements Comparable<Circle> {
```

...

```
public int compareTo(Circle o) {  
    double diff = this.getArea() - o.getArea(); // casting is not needed  
    if (diff > 0)  
        return 1;  
    else if (diff < 0)  
        return -1;  
    else  
        return 0;  
}
```

```
// triggers a compilation error if o is not passed a circle  
new ComparableCircleWithGeneric(1).compareTo("RED");
```

ComparableCircleWithGeneric.java

Raw Type is Unsafe - 2



```
interface Comparable<E> {  
    public int compareTo(E o);  
}
```

```
public class Max {  
    public static Comparable max(Comparable o1, Comparable o2) {  
        if (o1.compareTo(o2) > 0)  
            return o1;  
        else  
            return o2;  
    }  
}
```

■ `Max.max("Welcome", 23);` *// runtime error!*

□ *String cannot be compared with an integer*

Max.java

Raw Type is Unsafe - 2

```
interface Comparable<E> {  
    public int compareTo(E o);  
}
```

```
public class Max {
```

```
    public static Comparable max(Comparable o1, Comparable o2) {
```

```
        public static <E> Comparable max(E o1, E o2) { // ?
```

```
        public static <E extends Comparable> E max(E o1, E o2) { // ?
```


bounded type parameter

```
        public static <E extends Comparable<E>> E max(E o1, E o2) { // ?
```

■ `Max.max("Welcome", 23);` *// compilation error!*

□ *Goal: Make it triggers a compilation error*

Make it safe using bounded type parameter

- `<E extends Comparable<E>>` declares a bounded type parameter

```
public class Max1 {  
    public static <E extends Comparable<E>> E max(E o1, E o2) {  
        if (o1.compareTo(o2) > 0)  
            return o1;  
        else  
            return o2;  
    }  
}
```

```
interface Comparable<E> {  
    public int compareTo(E o);  
}
```

- `Max.max("Welcome", 23);` // compilation error!

[Max1.java](#)

Bounded Type Parameter

Task: Write an `equalArea` method that can compare two geometric objects.
Q: Which implementation will work?

```
public static void main(String[] args) {  
    Rectangle rectangle = new Rectangle(2, 2);  
    Circle circle = new Circle(2);  
    System.out.println("Same area? " + equalArea(rectangle, circle));  
}
```

```
public static <E> boolean equalArea(E object1, E object2) {  
    return object1.getArea() == object2.getArea();  
}
```



```
public static <E extends GeometricObject> boolean equalArea(E object1, E object2) {  
    return object1.getArea() == object2.getArea();  
}
```

[BoundedTypeDemo.java](#)

Issues in Using a Generic Type

```
public static void main(String[] args ) {  
    GenericStack<Integer> intStack = new GenericStack<>();  
    intStack.push(1); // 1 is autoboxed into new Integer(1)  
    intStack.push(2);  
    intStack.push(-2);  
    print(intStack);  
}  
  
// Print objects and empty stack  
public static void print(GenericStack<Object> stack) {  
    while (!stack.isEmpty()) {  
        System.out.print(stack.pop() + " ");  
    }  
}
```

*Can we use the
parameterized type
GenericStack<Object>
as a polymorphic type?*

```
public class GenericStack<E> {  
    private java.util.ArrayList<E> list =  
        ...  
}
```

Why bounded wildcards?

Can we use
`GenericStack<Number>`
or `GenericStack<?>` as
a type?

```
public static void main(String[] args ) {  
    GenericStack<Integer> intStack = new GenericStack<>();  
    intStack.push(1); // 1 is autoboxed into new Integer(1)  
    intStack.push(2);  
    intStack.push(-2);  
    System.out.print("The max number is " + max(intStack));  
}
```

// Find the maximum in a stack of numbers

```
public static double max(GenericStack<Number> stack) {  
    double max = stack.pop().doubleValue(); // initialize max  
    while (!stack.isEmpty()) {  
        double value = stack.pop().doubleValue();  
        if (value > max) max = value;  
    }  
    return max;  
}
```

```
public class GenericStack<E> {  
    private java.util.ArrayList<E> list =  
    ...  
}
```

[WildcardNeedDemo.java](#)

Why bounded wildcards?

```
public static void main(String[] args ) {  
    var intStack = new GenericStack<Integer>();  
    intStack.push(1); // 1 is autoboxed into new Integer(1)  
    intStack.push(2);  
    intStack.push(-2);  
    System.out.print("The max number is " + max(intStack));  
}
```

*We can use
GenericStack<? extends Number> as a type.*

// Find the maximum in a stack of numbers

```
public static double max(GenericStack<? extends Number> stack) {  
    double max = stack.pop().doubleValue(); // initialize max  
    while (!stack.isEmpty()) {  
        double value = stack.pop().doubleValue();  
        if (value > max) max = value;  
    }  
    return max;  
}
```

```
public class GenericStack<E> {  
    private java.util.ArrayList<E> list =  
        ...  
}
```

[WildcardNeedDemo.java](#)

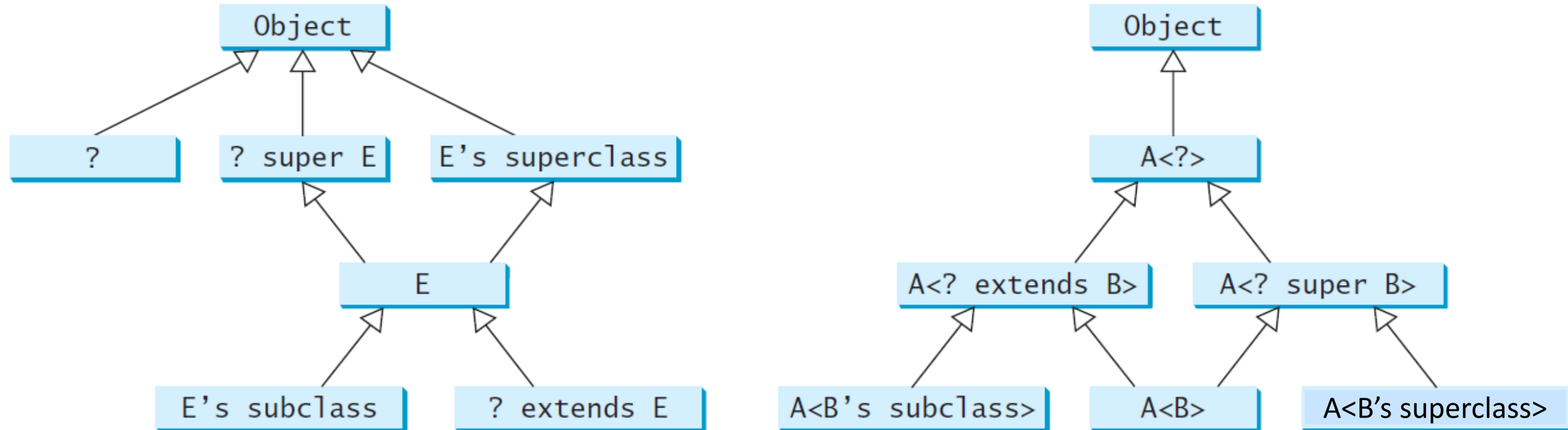
Terminology



- `GenericStack<? extends Number>` is a **bounded generic type**
- **`? extends Number`** is a **bounded type parameter**
- `GenericStack<? extends Number>` can be used as a **polymorphic type** so that its variables or method parameters can reference objects of multiple types in the previous `max()`

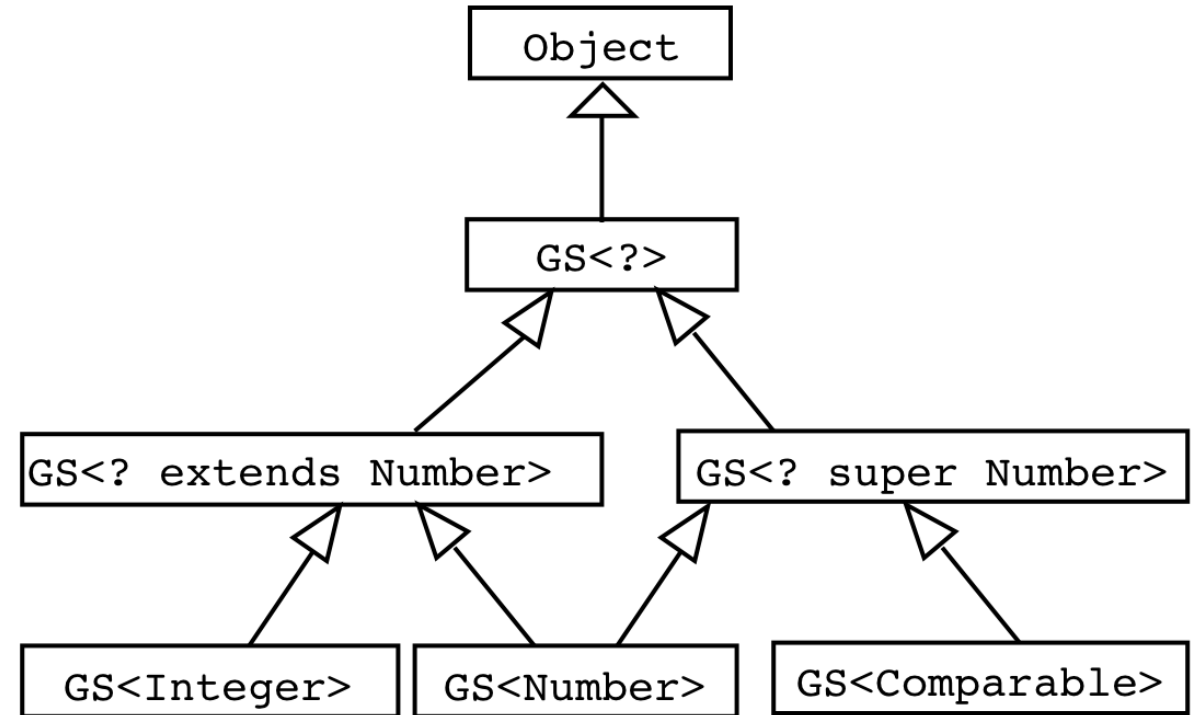
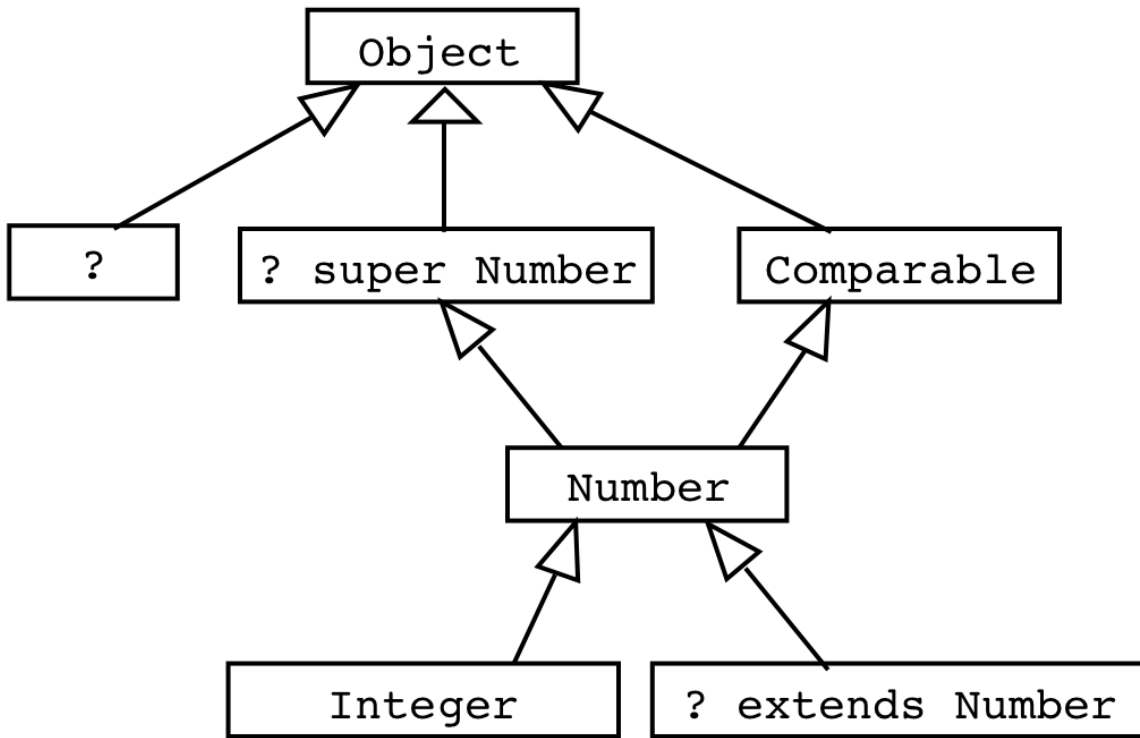
```
var intStack = new GenericStack<Integer>();  
var stringStack = new GenericStack<String>();  
...  
System.out.println("The max number is " + max(intStack));  
System.out.println("The largest string is " + max(stringStack));
```

Generic Types and Wildcard Types



Two type hierarchies defined for generic and wildcard types

Generic Types and Wildcard Types



Wildcard types can be used as values to bind a type parameter, such as `E` in a generic type `GS<E>`.

Exercise

```
public class TestPolymorphicType {  
    void check() {  
        Comparable<?> ca = "welcome";  
        Comparable<Object> co = null;  
  
        ✗ co = "hello";  
        ✗ co = (Comparable<Object>) "hello";  
        ✓ Comparable<? extends String> f = "hello";  
        ✓ ca = f;  
        ✗ f = ca;  
        ✗ co = ca;  
        ✓ co = (Comparable<Object>) ca;  
        ✓ ca = co;  
        ✓ Comparable c = ca;  
        ✗ Comparable<Comparable<?>> cc = co;  
        ✗ 🐛 Comparable<Comparable<?>> cc = (Comparable<Comparable<?>>) co;  
    }  
}
```

[TestPolymorphicType.java](#)

}

Erasure and Restrictions on Generics



- Generics are implemented using an approach called **type erasure**.
- The generic type information is removed after type checking.
- This approach enables the generics code to be backward-compatible with the legacy code that uses raw types.
 - E.g., Programs using ArrayList were not affected when ArrayList was changed to ArrayList<E> in Java 5.

Important Fact



- Note that a generic class is shared by all its instances regardless of its actual generic type.
 - `GenericStack<String> stack1 = new GenericStack<>();`
 - `GenericStack<Integer> stack2 = new GenericStack<>();`
- Although `GenericStack<String>` and `GenericStack<Integer>` are two types, there is only one class `GenericStack` loaded into the JVM.

Runtime cannot use generic type information



■ What will be printed?

```
public class TestInstanceof<E> {  
    public static void main(String [] args) {  
        var rawStack = new GenericStack();  
        var intStack = new GenericStack<Integer>();  
        System.out.println(intStack instanceof GenericStack<Object>);  
        System.out.println(intStack instanceof GenericStack);  
        System.out.println(intStack instanceof GenericStack<?>);  
        System.out.println(rawStack instanceof GenericStack<?>);  
    }  
}
```

compilation error because
the use of generic type is
not allowed at runtime



TestInstanceof

Restrictions on Generics

- Restriction 1: Cannot create an instance of a generic type. (i.e., `new E()`).
- Restriction 2: Generic array creation is not allowed. (i.e., `new E[100]`).
- Restriction 3: A generic type parameter of a class is not allowed in a static context.
- Restriction 4: Exception classes cannot be generic.

Lambda Expression

Functional programming in lambda expressions

- In Java, a **lambda (expression)** is an **anonymous function** to be performed by an object that implements a **functional interface**. We call the object a **function object**
- Mathematically, lambdas (a.k.a. closures) take the form:
 - $\lambda xy.t$ where x, y are variables and t is a lambda term
 - Example: $\lambda xy.x^2+y^2$
- In Java, a lambda expression takes the form:
 - $(x, y) \rightarrow t$ where x, y are variables and t is a lambda term
 - Example: $(x, y) \rightarrow x*x + y*y;$

Waiving two
paradigms into
one language

Type Inference of Lambda Expression

```
btUp.setOnAction(e -> {text.setY(text.getY() > 10 ? ...);});
```

1. Java compiler infers that the **argument's type** must be **EventHandler<ActionEvent>**
2. Java compiler infers that the concerned lambda expression must define the abstract method, which is **handle(ActionEvent evt)** in the interface

```
interface EventHandler<ActionEvent> {  
    public abstract void handle(ActionEvent evt);  
}
```

3. Java compiler infers that **e** in the lambda expression must corresponds to **evt** in **handle's** method parameter
4. Java compiler infers that the type of **e** in the lambda expression is **ActionEvent**, and **{text.setY(text.getY() > 10 ? ...);}** defines the overriding handle method's body

From Anonymous Innerclass to Lambda

■ Anonymous innerclass before Java 8

```
new Thread( new Runnable() { // anonymous inner class
```

```
    @Override
```

```
    public void run() {
```

```
        processSomeImage(imageName);
```

```
    }
```

```
});
```

()->processSomeImage(imageName)

is a lambda that creates a Runnable instance. We call the instance a function object because it is an object of a class that implements a functional interface. **A lambda is treated as a function object.**

■ Lambda since Java 8

```
new Thread(() no argument -> processSomeImage(imageName));
```

function object

Higher-Order Functions



- A higher-order function is a function that takes other functions (i.e., function objects) as arguments

```
interface Func {  
    public double eval(double x);  
}  
  
public class LambdaTest {  
    public double calc(Func f, double x) {  
        return f.eval(x);  
    }  
}
```

We provide a lambda
that defines f.eval()
each time we call calc()

Examples of using calc():

```
System.out.println(calc(x->x+x, 3));  
System.out.println(calc(x->x*x, 3));  
System.out.println(calc(x->x/x, 3));
```

Output:

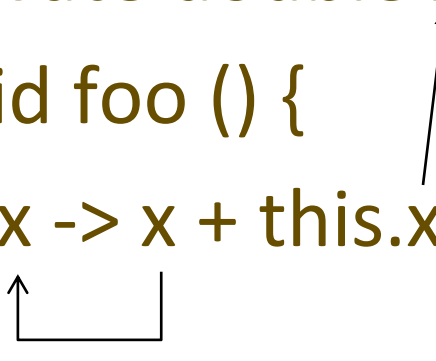
6.0
9.0
1.0

[LambdaTest.java](#)

Scope of Lambdas

Is the following code legal? What does this refer to?

```
class MyClass {  
    private double x = 0;  
    void foo () {  
        f(x -> x + this.x);  
    }  
}
```



Summary:

- The scope of lambda variables is their embedding method (e.g. foo() in the example).
- A lambda can access local variables and all variables/methods defined in its outer class
- Local variables used by lambda must be effectively final.
- “this” in a lambda (function object) refers to the instance of its embedding method. No separate .class files are generated for lambdas.

Legal: lambda variable name matches instance variable name

Built-in Functional Interfaces: Predicate<T>



■ java.util.function.Predicate<T>

```
public interface Predicate<T> {  
    boolean test(T t);  
}
```

Pretty simple
Is it useful?



- For creating a function object to **test if a condition holds**
- Commonly used to filter from a list (or stream) those elements that satisfy a condition. The condition test can be provided on-the-fly by a function object of type Predicate

Built-in Functional Interface: Function<T,R>



- Helps apply the function `apply(T t)` iteratively on each element of type **T** in a list with a result of type **R**

```
public interface Function<T,R> {  
    R apply(T t);  
}
```



If a functional interface has more than one type parameter, the last one tends to describe the output type and the remaining describes the lambda variable(s) type(s)

- Example: We apply a function to raise each employee's salary. The function takes an instance of **UStEmployee (T)** and returns an instance of **Double (R)**

```
Function<UStEmployee, Double> raise = e -> e.getSalary()*1.1;
```

```
for (var e: employees)  
    e.setSalary(raise.apply(e));
```

*raise references a function object that implements
apply(T t) using the lambda expression*

Built-in Functional Interface: Consumer<T>



- `Function<T,R>` assumes a function that returns a value of type `R`.
- What if a functions that returns nothing but introduces side effect to the input object `t`?
- `Consumer<T>`: Let us make a “function” that takes an object of type `T` and does some side effect to it (with no return value)

```
public interface Consumer<T> {  
    public void accept(T t); // makes side effect to t  
}
```

Type Casting of Functional Interfaces



■ functional interfaces

```
interface Consumer<T> {void accept(T t);}
interface MyInterface {void processEmployee(USTEmployee e);}
```

■ incompatible types cannot be automatically converted

```
Consumer<USTEmployee> raise = e -> e.setSalary(e.getSalary()*1.1);
MyInterface mi = e -> e.setSalary(e.getSalary()*1.2);
mi = raise; // error: types are not compatible
```

```
mi = (MyInterface) raise; // Can we cast Consumer<USTEmployee> to MyInterface?
```

Type Casting of Functional Interfaces



[LambdaEmployee.java](#)

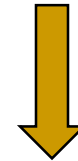
[TypeCasting.testTypeCasting\(\)](#)

■ functional interfaces

```
interface Consumer<T> {void accept(T t);}
interface MyInterface {void processEmployee(USTEmployee e);}
```

■ Cannot resolve setSalary and getSalary. Why?

```
var o = e -> e.setSalary(e.getSalary()*1.1); // error
```



Tell java compiler the lambda's type

```
var o = (MyInterface) e -> e.setSalary(e.getSalary()*1.1);
```

```
var o = (Consumer<USTEmployee>) e -> e.setSalary(e.getSalary()*1.1);
```

Function Composition



- So far, we pass non-function objects (e.g., USTEmployee) to the method defined in the built-in functional interfaces

```
public interface Predicate<T> {  
    boolean test(T t)  
}
```

```
// anonymous innerclass  
Predicate<USTEmployee> isYoung = new Predicate() {  
    boolean test(USTEmployee e) {  
        return e.getAge() < 30;  
    }  
}
```

- Predicate<USTEmployee> isYoung = **e** -> e.getAge()<30;
- Predicate<USTEmployee> isRich = **e** -> e.getSalary()>=30000;
- In both cases, we pass a USTEmployee object **e** to the test() method

Default Methods

- and, or, negate, isEqual are default methods defined by the built-in Predicate interface to facilitate predicate composition.
- They cannot be abstract methods because otherwise:
 - ❑ Predicate functional interface contains more than one abstract method, violating the language rule.
 - ❑ Developers need to implement these methods for each Predicate function object, which is tedious.
 - ❑ Java 8 provides default implementation of these methods.
 - ❑ and, or, negate can be overridden by Predicate function objects.

map



```
public static void main(String[] args) {  
    var numbers = Arrays.asList(1,2,3,4,5,6,7,8,9,10);  
    applyMap(numbers, v->v*2);  
}
```

```
public static <T, R> void applyMap(List<T> list, Function<T, R> f) {  
    list.stream()  
        .map(f)  
        .forEach(System.out::println);  
}
```

StreamOperation.java
applyMap

filter



```
List<Integer> numbers = Arrays.asList(1,2,3,4,5,6,7,8,9,10);  
numbers.stream()
```

```
    .filter(v -> v%2==0)
```

```
    .forEach(System.out::println);
```

StreamOperation.java
applyFilter

```
List<String> names = Arrays.asList("Bob", "Tom", "Jeff", "Jennifer", "Steve");  
final Function<String, Predicate<String>> startsWithLetter =
```

```
    letter->name->name.startsWith(letter);
```

```
names.stream()
```

```
    .filter(startsWithLetter.apply("J"))
```

```
    .forEach(System.out::println);
```

```
public interface Function<T,R> {  
    R apply(T t);  
}
```

<T> reduce(**base**, (T,T) -> T)



base value to be returned for an empty stream

StreamOperation.java
applyReduce



```
List<Integer> numbers = Arrays.asList(1,2,3,4,5,6,7,8,9,10);  
int sum = numbers.stream().reduce(0, (i, j)->i+j);  
System.out.println(sum);
```

```
List<String> names = Arrays.asList("Bob", "Tom", "Jeff", "Jennifer", "Steve");  
String longestName = names.stream().reduce("", (name1, name2) ->  
    name1.length() >= name2.length() ? name1 : name2);  
System.out.println(longestName);
```

```
double totalSalary = employees.stream().map(e->e.getSalary()).reduce(0.0, (s1, s2)->s1+s2);  
System.out.println("Total salary = " + totalSalary);
```

anyMatch(), allMatch(), count()



```
boolean anyMatch = employees.stream().anyMatch(e -> e.getAge()>42);  
System.out.println("Is there an employee older than 42? "+anyMatch);
```

```
boolean allMatch = employees.stream().allMatch(e -> e.getAge()>42);  
System.out.println("Are all employees older than 42? "+allMatch);
```

```
long n = employees.stream().filter(e -> e.getAge()>42).count();  
System.out.printf("There are %d employees older than 42.\n", n);
```

StreamOperation.java

Default Methods



```
interface Vehicle {  
    default void print() {  
        System.out.println("I am a vehicle.");  
    }  
}
```

```
class Car implements Vehicle {  
    void foo() {  
        print();  
    }  
}
```

- Default methods **must** be public.
- Default methods **cannot** be final.

Default Methods



```
interface Vehicle {  
    default void print() {  
        System.out.println("I am a vehicle.");  
    }  
}
```

```
class Car1 implements Vehicle {  
    public void print() {  
        System.out.print("I am a car and ");  
        Vehicle.super.print();  
    }  
}
```

- Default methods can be overridden.
- Default method can be called using **InterfaceName.super**.

Further Readings

Want to be a competent Java programmer?

- Documentation
- Reflection
- Collections
- Security
- Unit testing
- Internationalization
- Networking
- JDBC database access
- Object design patterns
- Spring framework
- COMP5111 (Spring 2021)

COMPETENCE



Resources:

<https://docs.oracle.com/javase/tutorial/>

<https://www.tutorialspoint.com/java/index.htm>

https://www.tutorialspoint.com/design_pattern/index.htm

<https://www.tutorialspoint.com/spring/index.htm>

Reviewing Quiz

Java Class and Main Method

Which of the following is/are correct?

- A source file cannot contain more than one class
- ✓ ■ A source file may contain no classes
- ✓ ■ We cannot declare more than one class to be public in a source file
- ✓ ■ A source file may contain no public classes
- ✓ ■ A public class does not necessarily define a main method
- ✓ ■ A non-public class may define a public main method
- ✓ ■ A class may define multiple main methods
- ✓ ■ The class that defines the public static void main(String[]) must have the same name as the source file
- ✓ ■ The body of a public class can be empty
- ✓ ■ We can only launch and run the method with the signature "public static void main(String [])"

Instance Variables

```
public class Test { private int i; private static final int j = 0;  
    public static void main(String[] args) { <stmt> } }
```

Consider the code above, which of the following is/are legal substitution of <stmt>?

- `i = 1;`
- `int j = i;`
- ✓ ■ `var t = new Test();`
- ✓ ■ `Object t = new Test();`
- ✓ ■ `var t = new Test().i;`
- ✓ ■ `var t = new Test().j;`
- ✓ ■ `int i = j;`
- `var t = new Test(0);`

Public Members in Non-public Classes

Which of the following about the program on 2-Classes.pdf (#28) is/are true?

- Compilation error occurs at Prototype because a non-public class cannot have public members.
- ✓ ■ Compilation error occurs at the import statement because Prototype cannot be imported to another package.
- Compilation error does not occur at the import statement but at Consumer.main method.
- ✓ ■ Compilation error occurs at the printf statement because Prototype cannot be resolved.
- No compilation errors found but the main method crashes when it runs.
- No compilation errors found and the main method outputs the HASHKEY value to console.
- None of the above.

Array Default Initialization

Which of the following is/are correct for the statement below?

```
var ary = new String [10];
```

- ✓ ■ It declares a variable ary that references a String array.
- It declares a variable ary of type String.
- ✓ ■ It declares a variable ary of type String[].
- It creates 10 String objects.
- ✓ ■ It creates an array object.
- ✓ ■ The array object created contains 10 elements with a null value.
- ✓ ■ The array object created contains 10 elements, each can reference a String object.

Enhanced For Loop

Which of the following about the code below is/are correct?

1: `var carr = new Circle[10];`

2: `for (var c: carr) c = new Circle();`

3: `for (int i = 0; i < carr.length; i++) carr[i] = new Circle();`

- ✓ ■ Line 1 creates an array instance whose elements can reference a Circle instance.
- ✓ ■ Line 2 creates 10 circle instances.
 - After executing Line 2, each element in carr references a circle instance.
- ✓ ■ Line 3 creates 10 circle instances.
- ✓ ■ After executing Line 3, each element in carr references a circle instance.

Immutability

Which of the following about the immutability below is/are correct?

- ✓ ■ The content of an immutable object cannot be changed once it is created.
- A class that is declared final is immutable.
- An immutable class cannot have public instance/static variables.
- ✓ ■ An object contains no variables is immutable.
- A class is immutable if no variables are declared in its class definition.
- ✓ ■ An immutable class provides no mutator (setter).
- ✓ ■ An immutable class provides no accessor (getter) that returns a reference to a mutable object.

Declaring an Array

Which of the following statements create an instance of an array?

- `float fa = new float[20];`

- `char[] ca = "Hello";`

- `var s = "Hello";`

- ✓ ■ `int[] ia = new int [15];`

- ✓ ■ `int[] ib = {0, 1, 2, 3, 4};`

String

What is the output of the program?

```
public static void main(String args[]) {  
    var a = "Hello";  
    var b = new String("Hello");  
    var c = "Hello";  
    System.out.print(a.equals(b));  
    System.out.print(a==b);  
    System.out.print(a==c);  
}
```

- true true true
- false false false
- true false true ✓
- true false false
- None of the above

Equals - 1

```
class Coordinate {  
    int x, y;  
    @Override  
    // define an overriding equals method  
}
```

Which of the following implementation of the equals method is/are legal?

- `boolean equals(Coordinate o) { return x == o.x && y == o.y; }`
- `public boolean equals(Coordinate o) { return x == o.x && y == o.y; }`
- `boolean equals(Object o) { return x == o.x && y == o.y; }`
- `public boolean equals(Object o) { return x == o.x && y == o.y; }`
- `public boolean equals(Object o) { if (o instanceof Coordinate c) return x==c.x && y==c.y; }`
- ✓ ■ None of the above

Equals - 1

```
class Coordinate {  
    int x, y;  
    @Override  
    public boolean equals(Object o) {  
        if (o instanceof Coordinate c)  
            return x == c.x && y == c.y;  
        else  
            return false;  
    }  
}
```

Equals - 2

Suppose `c` and `d` reference two `Coordinate` instances such that `c.equals(d)` is true. Which of the following **MUST** be true?

- `c == d`
- `c.hashCode() == d.hashCode()`
- `c.toString() == d.toString()`
- ✓ ■ `c.equals(d)`, where `var o = (Object) c;`
- None of the above

Constructor and Superclass - 1

```
class Parent { public Parent() { System.out.println("ABC"); }  
class Child extends Parent { }
```

Which of the statements is/are correct?

- Parent() is inherited by Child from Parent.
- ✓ ■ “ABC” is printed at console when a Child object is created.
- ✓ ■ A constructor may invoke a static method.
- ✓ ■ A default constructor is generated for Child.
- A class must have at least one non-private constructor.

Constructor and Superclass - 2

Which of the statements is/are correct?

- We can directly invoke a public instance method in superclass's superclass using “super.super”.
- ✓ ■ We can use super to invoke a superclass constructor.
- ✓ ■ We can use super to invoke a superclass method.
- ✓ ■ We can use super to invoke a superclass variable.
- ✓ ■ We can directly invoke a public static method in superclass's superclass.

Constructor and Superclass - 3

```
class Person { int x = 1; Person(int i) {...} ... }
```

Which of the statement(s) is/are correct?

- We can declare Person(int i) to be static.
- ✓ ■ Person(int i) cannot call both super() and this() in its body.
- ✓ ■ A constructor may invoke another overloaded constructor using 'this' keyword.
- ✓ ■ Instance variable x is initialized after the completion of Object().
- ✓ ■ Object() is executed once and only once for the creation of each Person instance.

Object and inheritance

An object that has more than one form is referred to as?

- interface
- inheritance
- abstract class
- ✓ ■ polymorphism
- overloading

Object and inheritance

Which of the following can assign the variable x to 1 in main()?

- `x = 1;`
- `super.x = 1;`
- `this.x = 1;`
- `Parent.x = 1;`
- ✓ ■ None of the above

```
class Parent {  
    public int x;  
}  
class Child extends Parent {  
    public static void main(String arg[]) {  
        // code to assign x to 1  
    }  
}
```

`new Child().x = 1;`

Overriding

Which of the statement(s) is/are correct?

- ✓ ■ An instance method can be overridden in subclass.
- A static method can be overridden in subclass.
- An instance variable can be overridden in subclass.
- ✓ ■ A subclass instance can be referenced by a superclass variable, e.g., `Superclass s = new Subclass();`
- A public instance method can be overridden with a protected instance method in subclass.

Accessibility of Protected

Suppose C and D are public subclasses of a public class B. Which of the statements is/are correct?

- ✓ ■ A protected method of B can be accessed by any classes that resides in the same package as B.
- ✓ ■ A constructor in Class C can access a protected constructor defined in B using 'super'.
- ✓ ■ Class C can access its inherited protected methods from B.
- ✓ ■ If C and B reside in different packages, C cannot access the protected methods inherited from B to D.
- ✓ ■ If C and B reside in the same package, C can access the protected methods inherited from B to D.

Overriding and Overloading

Which statement(s) is/are correct?

- ✓ ■ A method can be overloaded by another method in the same class.
- We can overload a static method by an instance method using the same parameters but different return types.
- ✓ ■ Two overloaded methods may have different accessibility.
- A private constructor cannot be overloaded.
- A method that declared to be final cannot be overloaded.

Instance_INITIALIZER Block

Which statements are correct?

- Field initializers are always executed before instance initializer blocks.
- Instance initializer blocks can use “this” to invoke a constructor of the current object.
- Instance initializer blocks can be inherited.
- We can declare different accessibility of instance initializer blocks using public, protected and private.
- ✓ ■ Instance initializer blocks are executed once in an object’s life time.
- ✓ ■ Instance initializer blocks may initialize superclass instance variables.

Static Initializer Block

Which statements are correct?

- A class can use “super” in its static initializer blocks to assign values to its superclass static variables.
- Like static methods, static initializer blocks can be inherited but cannot be overridden. E.g., `class B extends A { ... }`
- Static initializer blocks are executed once in an object’s life time.
- ✓ ■ A class may use its static initializer blocks to assign values to its subclass public static variables.
- None of the above.

Abstract Classes

Which of the following is/are legal?

- `class A { abstract void compute() { } }`
- `class A { abstract void compute(); }`
- ✓ ■ `public abstract class A { abstract void compute(); }`
- ✓ ■ `abstract class A { private void compute() { } }`
- ✓ ■ `abstract class A { }`
- `abstract class A { abstract final void compute(); }`

Abstract Classes

Which of the following is/are correct?

- ✓ ■ A subclass of a non-abstract superclass can be abstract.
- ✓ ■ A concrete method can be overridden by an abstract method.
- ✓ ■ An abstract class can be used as a data type to declare variables.
- Abstract classes cannot have default constructors.
- An abstract class may contain an abstract private method.
- An abstract class may contain an abstract static method.

Interfaces

Suppose A is an abstract class that implements interface F.
Which of the following is/are correct?

- `F o = new F();`
- `F o = new A();`
- ✓ ■ `F[] o = new F[10];`
- ✓ ■ `F[] o = new A[10];`
- ✓ ■ `Object o = new F[10];`

Interfaces

Which of the statements is/are legal?

- class B { }
- interface F extends B { } // wrong because interfaces cannot extend a class
- ✓ ■ interface F { void f(); }
- abstract class B implements F { }
- interface F { int i; } // wrong because constant needs to be initialized when it is declared
- interface F { void f(); }
- abstract class B implements F { void f() {} } // wrong because the accessibility is reduced
- ✓ ■ interface F { void f(); }
- abstract class B implements F { public abstract void f(); }

Anonymous Inner Class

Which of the following is/are correct?

- ✓ ■ Methods of an anonymous inner class can access the private instance fields and methods of its outer class.
- ✓ ■ An anonymous inner class can access the private static fields and methods of its outer class.
- ✓ ■ We cannot define a constructor explicitly for anonymous inner class.
- ✓ ■ We can define instance variables for an anonymous inner class and initialize them using instance initializer blocks.
- ✓ ■ An anonymous inner class that implements an interface cannot take any parameters because `java.lang.Object` supports only a no-args constructor.

Anonymous Inner Class

Which of the following is/are correct?

- ✓ ■ An anonymous inner class can access local variables of its host method only if they are effectively final.
- ✓ ■ An anonymous inner class can define its own static variables provided that they are final.
- An anonymous inner class can implement multiple interfaces.
- An anonymous inner class can contain abstract methods.
- An anonymous inner class can be used as a type.

Exception

Which of the statements is/are correct?

- All exceptions in Java must be checked.
- RuntimeException is a checked exception.
- A method must catch and handle the checked exceptions thrown by its callee methods.
- ✓ ■ It is impossible to catch exceptions thrown by statements executed outside a try block.
- ✓ ■ A method that throws a checked exception to its caller must declare so at its method signature using the “throws” keyword.

Exception

class A extends Exception {}

class B extends A {}

Which of the statements is/are correct?

- The exception handler of A can occur before the handler of B.
- We can use "catch (Object e) {...}" to catch A and B exceptions.
- We can use "catch (B e) {...}" to catch an A exception.
- ✓ ■ We can use "catch (A e) {...}" to catch a B exception.
- ✓ ■ In try-catch-finally, the finally block will always be executed even the try block contains a return statement.

Exception

Which of the following implementation of f() in Sub is valid?

```
class MyIOException extends IOException {}
```

```
class Super { void f() throws IOException {} }
```

```
class Sub extends Super {
```

```
    void f() throws Exception {}
```

```
    void f() throws ClassNotFoundException {}
```

```
    ✓ void f() throws IOException {}
```

```
    ✓ void f() throws MyIOException {}
```

```
    ✓ void f() {}
```

```
    ✓ void f() throws RuntimeException, Error {}
```

```
}
```

Exception and Interfaces

Which of the following is a valid implementation of clone() in MyClass?

```
interface Ifc extends Cloneable { Object clone(); }  
class MyClass implements Ifc {  
    @Override  
    // clone() implementation  
}
```

- public Object clone() { return super.clone(); }
- public Object clone() throws CloneNotSupportedException { return super.clone(); }
- public Object clone() {try{return super.clone();} catch(Exception e){ }}
- ✓ ■ public Object clone() {try{return super.clone();} catch(Exception e){ } return new Object(); }
- ✓ ■ public Object clone() throws RuntimeException { try{return super.clone();} catch(Exception e) {return new Object();} }

Exception and Interfaces

Which of the statements is/are legal?

- interface A extends Cloneable throws CloneNotSupportedException { public Object clone(); }
- ✓ ■ interface A extends Cloneable { public Object clone(); }
- ✓ ■ interface A extends Cloneable { public Comparable clone() throws CloneNotSupportedException; }
- ✓ ■ interface A extends Cloneable { public Comparable<A> clone() throws CloneNotSupportedException; }
- ✓ ■ interface A extends Cloneable { public Comparable<Cloneable> clone() throws CloneNotSupportedException; }

Text I/O

Which expression can be used to create an input object for file temp.txt?

- `new Scanner("temp.txt");`
- `new Scanner(temp.txt);`
- ✓ ■ `new Scanner(new File("temp.txt"));`
- `new Scanner(File("temp.txt"));`
- None of the above.

Text I/O

```
Scanner input = new  
Scanner(System.in);  
var v1 = input.nextInt();  
var v2 = input.nextInt();  
var line = input.nextLine();
```

Suppose we enter 34.3 57.8 789 followed by the ENTER key.

- After the last statement is executed, v1 is 34.
- The program has a runtime error because 34.3 is not an integer. ✓
- After the last statement is executed, line references a string of characters '7', '8', '9' and '\n'.
- After the last statement is executed, line references a string of characters '7', '8' and '9'.

Unit Testing

Which of the statements is/are correct?

- ✓ ■ A Unit Test method is characterized by a known input and an expected output, which is worked out before test execution.
- ✓ ■ A Unit Test method may test multiple operations of the Class under Test.
- ✓ ■ Annotating a public void method with `@AfterEach` in a Unit Test Class causes the method to be run after each Test method.
- ✓ ■ A Unit Test method should contain at least one assert statement to compare the actual result against the expected result.
- ✓ ■ JUnit provides a framework for automated test execution without human intervention.

Unit Testing

Which of the statements is/are correct?

- ✓ ■ A public void method annotated by `@BeforeAll` in a Test Class is executed once before any test methods in the class.
- ✓ ■ Testing is a process of checking if an application works according to its requirements.
- ✓ ■ A test method annotated with `@Ignore` in a Test Class will not be executed.
- ✓ ■ A test suite bundles a collection of unit tests and run them together.
- ✓ ■ When JUnit finds the current test method fails, it continues to run the next test method.

ArrayList and Accessibility

Which of the following is/are legal?

`ArrayList<Object> oList;`

`ArrayList<String> sList;`

- ✓ ■ `oList = new ArrayList<>();` .
- ✓ ■ `sList = new ArrayList<>();`
- `sList = new ArrayList<>(); oList = sList;`
- `sList = new String[10];`
- `String [] sArray = new ArrayList<>();`

JavaFX

Which of the statements is/are correct?

- ✓ ■ JavaFX is introduced to substitute Swing and AWT in Java 8.
- ✓ ■ Java compiler can still support programs written in Swing and AWT.
- ✓ ■ JavaFX is designed to provide multi-touch and built-in 3D support.
- ✓ ■ We can create more than one stage in an JavaFX application.
- ✓ ■ To write a GUI application, we define a class extending JavaFX Application.

JavaFX

Which of the statements is/are correct?

- Programmers need to create an instance of JavaFX Application before starting a GUI application.
- ✓ ■ To program a JavaFX application, we override its start() method.
- ✓ ■ A JavaFX application is launched using the Java Launcher thread.
- ✓ ■ JavaFX runtime uses the Launcher thread to call the init() method of a JavaFX application.
- ✓ ■ JavaFX runtime creates an application thread to run each JavaFX application.

JavaFX

Suppose a class MyApp is written to implement a JavaFX GUI application. Which of the statements is/are correct?

- ✓ ■ MyApp must extend `javafx.application.Application`.
- ✓ ■ MyApp must be a public class.
 - MyApp must not explicitly define any constructors.
 - MyApp must have only public constructors.
- ✓ ■ MyApp must override `start()` method.
- ✓ ■ MyApp must not call `super.start()` in the body of the overriding `start()` method.

JavaFX

What is the execution sequence of the following JavaFX program?

```
public class MyApp extends Application {  
    public MyApp() { ... }  
    public void start(Stage primaryStage) { ... }  
    public static void main(String[] args) { launch(args); }  
}
```

- start -> MyApp constructor -> launch
- start -> launch -> MyApp constructor
- MyApp constructor -> start -> launch
- MyApp constructor -> launch -> start
- launch -> start -> MyApp constructor
- ✓ ■ launch -> MyApp constructor -> start

JavaFX

Which statement(s) can add a node to a pane?

- `pane.add(node);`
- `pane.addAll(node);`
- ✓ ■ `pane.getChildren().add(node);`
- ✓ ■ `pane.getChildren().addAll(node);`
- `pane += node;`

Event Handling

Which of the following is/are correct?

- ✓ ■ `EventHandler<T extends Event>` is a generic interface containing an abstract method `public void handle(T event)`.
- ✓ ■ `EventHandler<ActionEvent>` is an instantiation of `EventHandler<T>`.
- ✓ ■ To set a *handler* to listen to the action events from a *source*, use `source.setAction(handler)`.
- ✓ ■ To add a *handler* to listen to the action events from a *source*, use `source.addEventHandler(ActionEvent.Action, handler)`.
- ✓ ■ We can set a handler to listen to events from multiple event sources.

Multithreading

```
public class Test implements Runnable {  
    public static void main(String[] args) { Test t = new Test(); }  
    public Test() {  
        // Which of the following can display "test" at console?  
    }  
    public void run() { System.out.println("test"); }  
}
```

- Thread t = new Thread(this); t.run();
- Thread t = new Thread(); t.start();
- Thread t = new Thread(); t.run();
- ✓ ■ Thread t = new Thread(this); t.start();
- ✓ ■ this.run();

Test.java

Multithreading

```
public static void main(String[] args) throws InterruptedException {  
    Thread t = new Thread(()->{System.out.println("Thread finished");  
    t.start(); t.sleep(5000);  
    System.out.println("Main finished");  
}
```

- Compilation error because main cannot throw exception.
- Runtime error occurs when executing t.sleep(5000).
- Thread t is put to sleep for 5 seconds after it starts. After it wakes up, "Thread finished" is printed, immediately followed by "Main finished".
- Thread t is put to sleep for 5 seconds after it starts. "Main finished" is printed. After t wakes up, "Thread finished" is printed.
- ✓ ■ Thread t prints "Thread finished" immediately after it starts. After 5 seconds, "Main finished" is printed.

Test2.java

Exception and Interfaces

Which of the statements is/are legal?

- interface A extends Cloneable throws CloneNotSupportedException { public Object clone(); }
- ✓ ■ interface A extends Cloneable { public Object clone(); }
- ✓ ■ interface A extends Cloneable { public Comparable clone() throws CloneNotSupportedException; }
- ✓ ■ interface A extends Cloneable { public Comparable<A> clone() throws CloneNotSupportedException; }
- ✓ ■ interface A extends Cloneable { public Comparable<Cloneable> clone() throws CloneNotSupportedException; }

Multithreading

Which of the following is/are true?

- ✓ ■ `yield()` is a static method of the `Thread` class.
- ✓ ■ `sleep(long milliseconds)` is a static method of the `Thread` class.
- ✓ ■ A synchronized instance method acquires an intrinsic lock (monitor) on the object for which the method is invoked.
- ✓ ■ A synchronized static method acquires an intrinsic lock (monitor) on the class of the object for which the method is invoked.
- When a thread executes `yield()` inside a synchronized method, it releases the intrinsic lock (monitor) that it has acquired.

Multithreading

Which of the following method(s) is/are true?

- ✓ ■ `expr` in `"synchronized (expr) {...}"` must be evaluated to reference an object or a class reference.
- ✓ ■ `wait()`, `notify()` and `notifyAll()` must be placed inside a synchronized method or block.
- ✓ ■ A thread that executes `obj.wait()` releases the acquired intrinsic lock on object `obj` before entering the blocked state.
- ✓ ■ `wait()` is an instance method defined in the `Object` class.
- ✓ ■ `notify()` and `notifyAll()` are instance methods defined in the `Object` class.

Multithreading

Which of the following is/are correct?

- ✓ ■ When we await or signal a lock's condition in a try-catch block, we should always invoke its `unlock()` method in the finally clause.
- ✓ ■ A condition object must be associated with a lock.
- ✓ ■ To invoke methods on a condition, its associated lock must be obtained first.
- ✓ ■ Once we invoke the `await` method on a condition, the lock is automatically released. Once the condition is right, the thread re-acquires the lock and continues execution.
- The `signal` method on a condition causes the associated lock for the condition to be released.
- When a thread executes `sleep()`, it releases the (intrinsic) locks that it has acquired.

Generics

Which of the following is/are valid?

- ✓ ■ `Comparable< String > c = new String("abc");`
- ✓ ■ `Comparable< String > c = "abc";`
- ✓ ■ `Comparable c = new String("abc");`
- `Comparable< Object > c = new String("abc");`
- `Comparable< String > c = new Object();`

Generics

Which of the following creates a list to store integers?

- `ArrayList< Object > list = new ArrayList< Integer > ();`
- `ArrayList< Integer > list = new ArrayList< Object > ();`
- `ArrayList< Integer > list = new ArrayList< ? > ();`
- ✓ ■ `ArrayList< ? > list = new ArrayList< Integer > ();`
- ✓ ■ `List< Integer > list = new ArrayList< > ();`

Generics

Which of the following is/are legal?

- ✓ ■ abstract class A< E > implements Comparable< E > { }
public int compareTo(E o), where E has no relation with A or Comparable
- ✓ ■ abstract class A< E extends A > implements Comparable< E > { }
public int compareTo(E o), where E is either A or a subclass of A
- ✓ ■ abstract class A< E > extends Number implements Comparable< E > { }
- abstract class A implements Comparable< E extends A > { }
- abstract class A implements Comparable< > { }

Generics

Which of the following is/are legal?

- ✓ ■ `< E > void f(ArrayList< E > e) { }`
- ✓ ■ `< E > void f(ArrayList e) { }`
- ✓ ■ `< E extends Number > void f(ArrayList< E > e) { }`
 - `< E > void f(ArrayList< E extends Number > e) { }`
- ✓ ■ `< E extends Comparable< ArrayList< E > > > E f(ArrayList< E > e) { return null; }`

Generics

Which of the following is/are correct?

- ✓ ■ `ArrayList< Integer >` is a subtype of `ArrayList< ? extends Number >`.
- ✓ ■ `ArrayList< Number >` is a subtype of `ArrayList< ? extends Number >`.
- ✓ ■ `ArrayList< ? extends Number >` is a subtype of `ArrayList< ? >`.
 - `ArrayList< ? extends Number >` is a subtype of `ArrayList< Object >`.
 - `ArrayList< ? extends Number >` is a subtype of `< ? super Number >`.

Generics

Which of the following is/are correct?

- ✓ ■ Generic type information is present at compile time.
- ✓ ■ Generic type information is not present at run time.
- ✓ ■ We cannot create an instance using a generic class type parameter.
- ✓ ■ We cannot use generic type parameters in static context.
- ✓ ■ We cannot create an array using a generic class, e.g., `new ArrayList< Integer >[10];`

Lambda Expressions

Which of the following the lambda expression is/are correct?

() -> System.out.println("function object is called")

- It is illegal because lambda expression takes at least one parameter.
- It can be used to extend an abstract class containing one abstract method.
- ✓ ■ The interface that it implements must contain one abstract method.
- ✓ ■ It can be an implementation for multiple interfaces.
- ✓ ■ We can assign it to a variable of type `java.lang.Object`.

Lambdas

```
public class LambdaTest {  
    public static void main(String[ ] args) {  
        // Which of the following statement(s) can be used here?  
    }  
    static int calc(Func f, int x) { return f.eval(x); } }  
interface Func { int eval(int x); }
```

- ✓ ■ System.out.println(calc(x -> x+x, 3));
- ✓ ■ System.out.println(calc(x -> x*x, 3));
- ✓ ■ System.out.println(calc(x->calc(y->y*y, x), 3));
- System.out.println(calc(() -> x/x, 3));
- ✓ ■ System.out.println(calc(x -> 3, 3));

Lambdas

Which of the following about the code is/are correct?

- Ifc is not a functional interface because it contains two abstract methods.
- A compilation error occurs at Impl because it has not defined the implementation of equals.
- The code compiles but the execution of **new Impl().equals("")** will trigger a runtime exception.
- equals in Ifc overrides the equals method in Object with an abstract method.
- ✓ ■ We can create an instance directly from Impl.

```
@FunctionalInterface
interface Ifc {
    void f();
    boolean equals(Object o);
}

class Impl implements Ifc {
    public void f() {}
}
```

Lambdas

```
public static void main(String[] args) {  
    String name = "Comp3021";  
    name = name.toUpperCase();  
    Runnable r = () -> System.out.print(name);  
    r.run();  
}
```



- The program cannot be compiled.
- The program runs and print "Comp3021" at the console.
- The program runs and print "COMP3021" at the console.
- The program compiles but nothing appears at console because r needs to be run by calling its start method.
- The program runs and creates a thread to run r.

THE END OF JAVA REVIEW



**SEE YOU IN FINAL EXAMINATION ON DEC 18
(FRIDAY), 16:30 ON CANVAS AND ZOOM**