EC327 – Practice Exam II (with Answers)

1 True or False – General Knowledge (14 pts, 2 pts each)

State True or False and give a one-sentence explanation.

1. A well-designed dependency injection system removes all need for unit testing.
   Answer: False. DI makes testing easier by decoupling, but it does not eliminate the need for unit tests.

2. Android's `onClickListener` callback runs on the main UI thread.
   Answer: True. All UI callbacks in Android are dispatched on the main thread.

3. Objects allocated with `new` on the stack must be explicitly `delete`-ed.
   Answer: False. `new` allocates on the heap; stack objects are automatic and never use `new`.

4. In Scrum, story points measure the actual hours a task will take.
   Answer: False. Story points measure relative effort, not calendar hours.

5. In C++, marking a method `override` changes its dispatch from static to dynamic.
   Answer: False. `override` enforces that a method overrides a virtual base; `virtual` controls dispatch.

6. Accessing Android UI widgets from a background thread is safe if you synchronize.
   Answer: False. The Android UI toolkit is not thread-safe; changes must occur on the main thread.

7. In object-oriented design, child classes should freely access parent private members.
   Answer: False. Private members are inaccessible to subclasses; use `protected` if needed.

2 Multiple Choice (12 pts, 2 pts each)

Choose the single best answer and provide the letter.

1. Which is *not* a benefit of using `std::vector` over raw arrays?

   (A) automatic resizing

(B) bounds checking on `at()`

(C) constant-time `push_back` amortized

(D) built-in garbage collection

Answer: D

2. In Kanban, how is work typically assigned?

(A) Manager pushes tasks to developers

(B) Developers pull tasks when ready

(C) Tasks are auto-assigned by the tool

(D) Tasks are planned six months in advance

Answer: B

3. What does the C++ `final` specifier on a virtual function do?

(A) Disables further overriding

(B) Forces dynamic dispatch

(C) Makes it inlineable

(D) Marks it as pure virtual

Answer: A

4. Which Android persistence option is best for storing small key-value pairs?

(A) SQLite via Room

(B) Files in internal storage

(C) SharedPreferences

(D) External SD card files

Answer: C

5. In a `std::map`, what is the complexity of `lower_bound(k)`?

(A) O(1)

(B) O(log N)

(C) O(N)

(D) O(N log N)

Answer: B

6. What's the primary difference between composition and inheritance?

(A) Composition reuses code, inheritance does not

(B) Inheritance models "has-a," composition models "is-a"

(C) Composition favors interfaces, inheritance builds hierarchies

(D) Inheritance is always preferable for code reuse

Answer: C

3 Templatized Class (12 pts)

3.1 Implementation (9 pts)

Declare and implement in `stack.h`:

```
\#ifndef STACK\_H
\#define STACK\_H

\#include <vector>
\#include <algorithm>

template<typename T>
class SimpleStack {
private:
std::vector<T> elements;
public:
void push(const T& value) { elements.push\_back(value); }
void pop() { elements.pop\_back(); }
T& top() { return elements.back(); }
bool empty() const { return elements.empty(); }
bool contains(const T& value) const {
return std::find(elements.begin(), elements.end(), value)
!= elements.end();
}
};

\#endif // STACK\_H
```

3.2 Complexity (3 pts)

- `push`: O(1) amortized
- `pop`: O(1)
- `contains`: O(N)

4 Inheritance & Dispatch (12 pts)

4.1 NotificationSender Base (3 pts)

```
// notificationsender.h
\#ifndef NOTIFICATION\_SENDER\_H
\#define NOTIFICATION\_SENDER\_H

\#include <string>

class NotificationSender {
public:
virtual void send(const std::string& msg) = 0;
virtual \~NotificationSender() = default;
};

\#endif
```

4.2 SMSSender Subclass (2 pts)

```
// smssender.h
\#ifndef SMS\_SENDER\_H
\#define SMS\_SENDER\_H

\#include "notificationsender.h"

class SMSSender : public NotificationSender {
public:
void send(const std::string& msg) override {
// ... send via SMS ...
}
};

\#endif
```

4.3 EmailSender Subclass (3 pts)

```
// emailsender.h
\#ifndef EMAIL\_SENDER\_H
\#define EMAIL\_SENDER\_H

\#include "notificationsender.h"
\#include <string>
```

```
class EmailSender : public NotificationSender {
private:
std::string\* server;
public:
EmailSender(std::string\* srv) : server(srv) {}
void send(const std::string& msg) override {
// ... send via \*server ...
}
\~EmailSender() override { delete server; }
};

\#endif
```

4.4 Runtime Behavior (4 pts)

Given:

```
auto\* srv = new std::string("smtp");
NotificationSender\* s = new EmailSender(srv);
s->send("hello");
delete s;
```

- **Output:** EmailSender's send logic runs (e.g. sending "hello" via SMTP).
- **Dispatch:** Virtual dispatch via v-table calls EmailSender::send.
- **Destructor:** ẼmailSender runs first (deleting server), then ÑotificationSender.

5 Standard Template Library (20 pts)

5.1 Declarations (8 pts)

```
\#include <string>
\#include \<unordered\_map>
\#include <map>
\#include <vector>

class User; // forward

class UserManager {
private:
std::unordered\_map\<std::string,const User\*> usersByName;
std::map\<int,std::vector\<const User\*>> usersByAge;
public:
```

```
bool registerNewUser(const User\*);
const User\* getUser(const std::string&) const;
const User\* getYoungestUserOlderThan(int) const;
};
```

5.2 Implementations (6 pts)

```
\#include "usermanager.h"
\#include "user.h"

bool UserManager::registerNewUser(const User\* u) {
auto \[it, inserted] = usersByName.emplace(u->getUsername(), u)
   ;
if (!inserted) return false;
usersByAge\[u->getAge()].push\_back(u);
return true;
}

const User\* UserManager::getUser(const std::string& name)
   const {
auto it = usersByName.find(name);
return it != usersByName.end() ? it->second : nullptr;
}

const User\* UserManager::getYoungestUserOlderThan(int age)
   const {
auto it = usersByAge.lower\_bound(age);
if (it == usersByAge.cend()) return nullptr;
return it->second.front();
}
```

5.3 Complexity Explanations (6 pts)

- **getUser O(1):** Unordered map lookup on hash table is average O(1).
- **getYoungestUserOlderThan O(log N):** `map::lower_bound` on red-black tree is O(log N), plus O(1) to return the first element.

6 Android Development (20 pts)

6.1 Code Analysis (6 pts)

6.1.1. Adds an `OnClickListener` to `addButton`; on click, reads text, updates list, and refreshes the `TextView`.

Answer: See description.

6.1.2. Must avoid long work in the click callback because it runs on the UI thread; blocking it causes ANR.
Answer: UI thread must remain responsive.

6.1.3. Yes. In-memory updates and cheap `TextViewsetText` keep 50 rapid additions smooth.
Answer: Operations are light and asynchronous persistence does not block UI.

6.2 Persistence (8 pts)

6.2.1. Strategy: Load once in `onCreate` from `SharedPreferences`; updates never saved by default.
Answer: Uses `prefs.getString` in `loadItems` only.

6.2.2. Shortcomings: `saveItems()` is never called, so disk never updated; data lost on process death.
Answer: No persistence of user actions.

6.2.3. Fix: Call `saveItems()` after each add or in `onPause()`.
Answer: Ensures `apply()` is invoked.

6.2.4. Alternatives: SQLite/Room (structured, schema), JSON/XML files (manual parsing), external storage (permissions/trade-offs).
Answer: Varying complexity and performance.

6.3 Server Sync (6 pts)

6.3.1. No: network I/O on UI thread throws `NetworkOnMainThreadException` and risks ANR.
Answer: Must offload network work.

6.3.2. Better: Use `ExecutorService` or background `Thread`, then post results via `runOnUiThread` or `Handler`.
Answer: Keeps UI responsive.

6.3.3. On success/failure callback, use `runOnUiThread` to show a Toast or update a status indicator.
Answer: UI update on main thread after background completion.

7 Polymorphism, Composition & Inheritance (10 pts)

Given the `Entity` interface and subclasses:

7.1 Design Questions (4 pts)

7.1.1. Does this support polymorphism?
Answer: Yes; you can treat any subclass as an `Entity`.

7.1.2. Are these relationships "is-a" or "has-a"?
Answer: *Is-a* (each subclass extends the base interface).

7.1.3. Is this composition or inheritance?
Answer: Inheritance (classes implement/extend `Entity`).

7.2 Alternative with Composition (2 pts)

Instead of inheritance, each `Player`, `Enemy`, etc. could *contain* a `BasicEntityImpl` and delegate calls:

```
class Player : public Entity {
private:
BasicEntityImpl impl;
public:
void move(int x,int y) override { impl.move(x,y); }
void render(GameScreen s) override { impl.render(s); }
Entity collide(Entity e) override { return impl.collide(e); }
};
```

7.3 Pros/Cons (3 pts)

- Composition:
  - + Better encapsulation, flexibility, and decoupling.
  - – More boilerplate delegation code.
- Inheritance:
  - + Less boilerplate, direct reuse of base logic.
  - – Tighter coupling, fragile base-class issues.

7.4 Dependency Injection (1 pt)

You could inject different implementations of collision or rendering logic (e.g., via constructor parameters) rather than hard-coding in subclasses.